UiO : Department of Informatics
University of Oslo

A Server-Side Feature Detection System for an Enterprise Web CMS

Master's Thesis
60 Credits

Henrik Hellerøy
Spring 2013

A Server-Side Feature Detection System for an Enterprise Web CMS

Master's Thesis
60 Credits

Henrik Hellerøy

SPRING 2013

Department of Informatics, University of Oslo

Abstract

Mobile devices have in later years become increasingly popular to use for browsing the Web. With this increased popularity, the limitations of the devices, along with the complexity of modern Web pages, have become apparent. Several methods have been introduced to alleviate the issues encountered on the "Mobile Web", including Responsive Web Design (RWD) and Device Experiences (DE). In this thesis, I present the Web technologies and concepts involved historically, along with their perceived problems today. Furthermore, I present an implementation of a system utilizing a third concept for improving performance on the Web: Responsive Web Design + Server Side Components (RESS), implemented as a server-side feature detection plugin for a Content Management System (CMS) named Enonic. This plugin strives to improve the CMS's future friendliness, as well as looking into the viability of RESS as a concept for improving "Mobile Web" performance. My results demonstrate that the plugin may provide a more flexible and future friendly method for creating Web sites that are responsive not just on the client side, but also on the server side. The results also show that the plugin achieves this without imposing a significant latency increase in HTTP requests made to the CMS.

The code for the implementation can be accessed from my GitHub repository at: https://github.com/helleroy/enonicDetector

Acknowledgements

This report has been written as a master's thesis for 60 credits at the Department of Informatics, University of Oslo. The work was performed in the period August 2012 to April 2013, under the supervision of Håvard Tegelsrud of Bekk Consulting AS and Dag Langmyhr from the Department of Informatics, University of Oslo.

I would like to thank Håvard for providing me with excellent advice and feedback whenever I needed throughout, for bringing his own expertise to help me complete the implementation as well as this report, and lastly for being an all-around fantastic person to work with! To Dag I extend my gratitude for taking the time to assist me with his extensive experience and providing much-needed quality control during the final hours of my work.

The thesis was written in cooperation with Bekk Consulting AS, to which I would like to express thanks for providing me with the resources and expertise needed throughout the whole process, from finding a problem statement and finalizing the thesis.

Finally I would like to thank my family and friends for all their love and support.

Table of Contents

1 Introduction

Can a Web site be made responsive without having the user agent do all the work?

1.1 Background

Modern Web sites contain a multitude of functionality and rich, interactive content. With this rich content and interactivity there comes a cost in the form performance issues. Mobile devices are quickly becoming peoples main channel for accessing content on the Web [1], much of which is either not designed for or is poorly implemented for use on such devices. Web applications are often highly optimized on the back end, i.e. on Web servers, for the sake of scalability through database tuning, clustering, customized data caching and so on, which allows them to handle a large number of requests. Although this performance tuning helps the applications service a large number of users, the users themselves do not experience these optimizations in any tangible manner. Users are interested in their own request, and if it is slow, the quality of the user experience is severely diminished.

With mobile devices comes a severe reduction in the amount of screen space Web developers can utilize to present content, along with reduced processing power and memory. Considering that Web pages now need to work in both desktop and mobile contexts, developers have started coming up with ways to simplify and streamline the process of creating them that adapts to the environment in which they are being viewed. The most popular among these is “Responsive Web Design” (RWD) – suggested in the book by the same name [2]. Roughly speaking it aims to make Web pages “respond” to the context in which they are being viewed. This is primarily achieved through something called “Media Queries” in CSS. Media Queries can be used to detect certain attributes of the device rendering the Web page, e.g. screen width and height, which then can alter its layout to fit the result of the query. It is not without its problems, though, and it has been noted that it is not a “silver bullet” for mobile Web design. Doing all of the adaptation on the front end causes file sizes and business logic to grow much

larger than previously, as all clients now receive the entire HTML markup, CSS, JavaScript and media [3].

1.1.1 Web performance

Steve Souders, author, creator of the Web browser performance plugin YSlow and engineer at Google, suggests that because of this we should focus on improving the response time on the front end, i.e. the Web browser. The front end, he says, stands for 80-90 percent of the response time [4]. He suggests a list of best practices aimed at improving the performance of Web pages through front-end optimization:

> *I set out to capture these best practices in a simple list that is easy to remember. The list has evolved to contain the following 14 prioritized rules:*
>
> 1. *Make fewer HTTP requests*
> 2. *Use a content delivery network*
> 3. *Add an Expires header*
> 4. *Gzip components*
> 5. *Put style sheets at the top*
> 6. *Put scripts at the bottom*
> 7. *Avoid CSS expressions*
> 8. *Make JavaScript and CSS external*
> 9. *Reduce DNS lookups*
> 10. *Minify JavaScript*
> 11. *Avoid redirects*
> 12. *Remove duplicate scripts*
> 13. *Configure ETags*
> 14. *Make Ajax cacheable*
>
> *- Steve Souders*

Many design philosophies and best practices have surfaced with the advent of the mobile Web; RWD is, as mentioned, one of the most popular among these. While this method is practical for developers in terms of giving them an easy way to make their Web pages adapt to their environment, it leaves the whole job of making the page adapt on the front end. This is somewhat contradictory to Souders' idea of optimizing the front end of Web applications, because that is where the largest chunk of the response time is spent in such a solution. RWD is reliant on heavy use of CSS and sometimes also needs to use JavaScript to hide and alter content to fit each device. Both of these things go against Souders' principles. Doing these kinds of alterations on the front end also does not

reduce the amount of HTTP requests, as these are defined in the HTML markup that is sent to the user from the back end.

On the opposite side of Souders we have people like Kate Matsudaira, a previous technical lead/manager at Amazon and Microsoft, who suggests that to improve the performance of the mobile Web we need to improve the back end [5]. She says that because of the limited system resources and bandwidth, we need to minimize connections and data across the network, images and other media by leveraging technologies such as LocalStorage (an HTML 5 technology for storing data on the client) and caching, as well as allowing the server to correctly identify the limits of the device making a request. While her article focuses on API design for the mobile Web, it touches upon an interesting question: how can we improve the detection capabilities of the server to improve Web performance?

1.1.2 Making the back end smarter

Jon Arne Sæterås, Mobile Web evangelist, blogger and product director at Mobiletech suggested in his post "Next steps of Responsive Web Design", that letting the device do all the hard work of being responsive is neither fair, right or smart [6].

> *It is not only the design of the web site and the layout of content that needs to be adapted of enhanced; the idea of being responsive, adaptive and enhancing, must be implemented in the whole value chain.*
>
> *– Jon Arne Sæterås*

In its current form, RWD sends the same markup, CSS, JavaScript and images to all devices, regardless of their capabilities. What he means by his statement is that more parts of the Web hierarchy (server, CMS, editor etc.) must be able to respond to the capabilities of the device making the HTTP request. Being able to move much of the work over to the back end can potentially reduce much of the current load on devices viewing Web pages designed using RWD. Involving the whole "value chain", as he calls it, will result in Web pages consuming less time and resources on the front end, leaving the responsibility of things like image scaling and markup processing to the back end.

Responsive Design + Server Side Components (RESS) is a method of making the server participate more actively in RWD. It suggests having Web pages split into components that are altered by the server depending on the type of user agent (UA) it detects as the requestor [7]. This idea goes along with Sæterås's notion of making more of the "value chain" smarter, in this case the Web server. It allows the server to more accurately tune what is sent to the client, amongst other things giving it the ability to optimize bandwidth usage when responding to mobile devices.

1.1.3 Future Friendly

Being able to make a Web site adapt to all present and future devices is a daunting proposition. Making it "future proof" is near impossible, as there is no way of telling what kind of devices people will be using to access the Web in the future. In 2011, several of today's biggest Web gurus got together and proposed a new Web design philosophy: be Future Friendly, not future proof [8, 9]. It suggests that being "future proof" is unnecessary (and near impossible), and that focusing on what actually brings value to your site will help it survive longer. It further proposes focusing on content first, orbiting around the data and having dynamic feature detection systems that do not rely on manual updates. Through following these principles, they claim, Web pages can be made ready to function on future devices without developers having to constantly maintain and update their pages.

1.2 Thesis description

In this thesis I present an implementation of a possible solution for making Web pages responsive without having all of the work done on the front end, based on the RESS concept and inspired by a RESS-like system called "Detector" made by Dave Olsen [10]. This is done with the aim of improving Web performance and the user experience on both desktop and mobile devices. The focus has been on the mobile aspect as this is where the biggest gains can be found in the context of improving performance, and thus the user experience. The implementation is a "server-side feature detection" plugin for the Enonic CMS, which aims to accurately detect the features supported by each individual UA. This is meant to allow the HTML document served to the user to be

tailored on the back end before ever reaching the requesting UA. The idea is that by tailoring the HTML on the back end, the performance on the front end can be improved, following the concepts of Souders and Matsudaira, amongst others, and making the Web pages built in the CMS more "Future Friendly".

1.2.1 Structure

The thesis is structured in a "bottom up" fashion to provide all necessary background knowledge before delving into the details of the implementation. The point of this is to draw a clear picture of what I worked with and why I did it, as well as providing a historical context to the problems developers face today.

Chapter 1 is meant to provide a quick introduction to the problem we are discussing and what I have implemented.
Chapter 2 presents the technologies and concepts that this thesis is based upon, as well as the related work in this field.
Chapter 3 presents the Enonic CMS, on which I have developed plugin – how it works and how it supports plugins.
Chapter 4 discusses the details surrounding the implementation of the plugin, what choices I made during the development process and why, as well as how I worked and what problems I encountered.
Chapter 5 details how I did the performance testing of the plugin and also presents the results of the tests.
Chapter 6 discusses the merits of the plugin, the results of the performance tests and attempts to look at the results in the context of related work in the same field. Finally I summarize the key findings and discuss whether or not the implementation was successful.

The thesis can be read in three parts: the first part (chapters 1-3) containing the background information needed to understand the implementation, the middle part (chapters 4 and 5) can be considered the practical part, describing the process of implementing the system and the performance tests conducted on it to establish the impact it might have on making the Enonic CMS' device classification system more

Future Friendly. Finally, the last part (chapter 6) is a discussion where we discuss the implementation, its merits and the results from the performance tests. It also looks at the concepts themselves and how they tie into the implementation.

2 Theory

In this chapter we are going to take a closer look at core Web technologies and concepts such as HTML, CSS, JavaScript, which are key to understanding the thesis. We will also look at technologies that are important in understanding device detection as a general concept, as well as the implementation of the server-side feature detection system, which we are going to discuss in later chapters. Related work is presented at the end of the chapter to provide an overview of similar projects and implementations.

2.1 Technologies

2.1.1 Hypertext Markup Language (HTML)

HTML is the main markup language for displaying Web pages and other information that can be displayed in a Web browser. The physicist Tim Berners-Lee first proposed it as an Internet-based hypertext system in 1989 while he was working as a contractor at CERN [11]. He wrote the browser and server software to run the system at the end of 1990 and the first publicly available description of HTML was released on the Internet by him at the end of 1992 [12]. This first document described 18 semantic elements comprising the language, many of which still exist in today's HTML standard. The language consists of preset tags that form elements with semantic meaning that Web browsers use to interpret the content of a Web page. This structure is based on SGML (Standard Generalized Markup Language), which is an ISO-standard technology for defining generalized markup languages [13]. These elements can have attributes that primarily exist as name-value pairs. Common attributes include "id", which defines a unique document-wide identifier for the element, and "class", which gives the ability to classify similar elements. This can be used for both semantic and presentational purposes.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello HTML</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

Code Snippet 1: A Minimal HTML document with a title and a paragraph of text.

While earlier HTML standards included presentational tags for manipulating the look of data in a document, they were deprecated in the HTML 4 standard of 1999 and made illegal in HTML 5, giving the responsibility of presentation to Cascading Style Sheets (CSS), and leaving HTML with giving documents structure and semantic meaning.

Web browsers interpret HTML and the document is rendered in a Document Object Model hierarchy (DOM) that defines the structure of a Web page. Furthermore, scripts embedded in the markup, such as JavaScript (JS), can manipulate this HTML DOM after it has been rendered to change the behavior and layout of a page – creating dynamic Web pages.

Today the World Wide Web Consortium (W3C) maintains the HTML standard. Though in the beginning HTML was largely shaped by various browser vendors who, for the most part, did what they pleased and added tags as they saw fit. Doing this led to the HTML standard containing many elements and attributes that are either deprecated or gone today, often because they mixed structure with presentation. To be backwards compatible with HTML-documents created in this period, today's browsers interpret markup in two different modes depending on the document type defined in the HTML: "Standards mode" is the regular parsing mode for modern browsers, where they demand adherence to the HTML standard defined in the document type at the top of the document, such as HTML 4.01 or HTML 5. If no valid document type is defined in a document a browser reverts to "Quirks mode" which is more lenient towards deprecated markup as well as attempting to fix markup that is flawed in some way – e.g. opening tags that do not have corresponding closing tags.

Different browsers interpret markup slightly differently, and some are more standards-compliant than others. This leads to developers having to write markup that is supported by all popular browsers, while using JS code to provide facilities that are not supported by less standards-compliant browsers, such code is know as a "polyfill". Using polyfills is common in modern Web pages and applications to provide support for HTML 5 features in browsers such as Internet Explorer (IE) 8 and older, as these do not support much of the HTML 5 standard. We will further elaborate the concept of polyfills later in this chapter.

Besides HTML there exists a stricter version called XHTML (Extensible Hypertext Markup Language) that is based on XML (Extensible Markup Language) instead of SGML. XML is a more restrictive subset of SGML, which is most commonly used to transfer data over the Internet in a format that is both human-readable and machine-readable. XHTML was created with the intention of creating a version of HTML that would be compatible with common XML tools, such as the XML parser, to fix the problems with different browsers interpreting the SGML standard, and thus the HTML standard, differently. XML demands that markup is well formed, meaning that it must adhere to strict syntactical rules that cannot be broken, such as case sensitivity, closing all tags in correct order and so on. Code Snippet 1 is an example of well formed markup. Being forced to follow these rules strictly will make sure that all documents are syntactically identical, and thus leave no room for misunderstanding between browsers. The XHTML standard is being developed along side the HTML standard.

2.1.2 Cascading Style Sheets (CSS)

CSS is responsible for describing the presentation semantics in documents written in markup languages. Most commonly it is used for styling Web pages written in either HTML or XHTML. Håkon Wium Lie, who at the time worked alongside Berners-Lee at CERN and currently works as chief technology officer at Opera Software, proposed it to the W3C in 1994 [14], which promptly added work on CSS to the deliverables of the HTML editorial review board. Lie designed it to solve the problem of having a consistent Web page appearance across browsers, as well as separating the structural semantics in HTML from presentational semantics.

CSS enables the separation of document content from document presentation, which can improve accessibility while providing more flexibility in the presentational characteristics of a Web page. It also provides the ability to have multiple pages share formatting through the use of the same style sheet – enabling an easy way to create a uniform presentational profile for Web pages.

The syntax of CSS is based on simple English words to define rule sets for various elements in a document described by "CSS selectors", which can be seen in Code Snippet 2. Selectors can reference either HTML elements directly, or "id" and "class" element attributes by prefixing attribute-specific characters. A pound sign (#) is prefixed to an "id" selector, and a period (.) is prefixed to a "class" selector. Each rule set consists of one or more properties that are arranged in name-value pairs, separated by semicolons. Rule sets can also have pseudo-classes appended to them that apply to information from outside the DOM hierarchy of the document, such as "*:hover"*, which applies to elements that a user hovers the mouse pointer over.

```
selector [, selector2, ...] [:pseudo-class] {
 property: value;
 [property2: value2;
 ...]
}
/* comment */
```

Code Snippet 2: A CSS rule set with two selectors, a pseudo-class, two properties and a comment.

To define the margin, border and padding of an element in the DOM tree, CSS generates several rectangular boxes around the element in what is called the “Box Model”. The Box Model, as seen in Figure 1, is defined as having the content of the element in the center, followed by padding, a border, and a margin. The padding spaces the content from its border, while the margin spaces the border from other elements in the DOM.

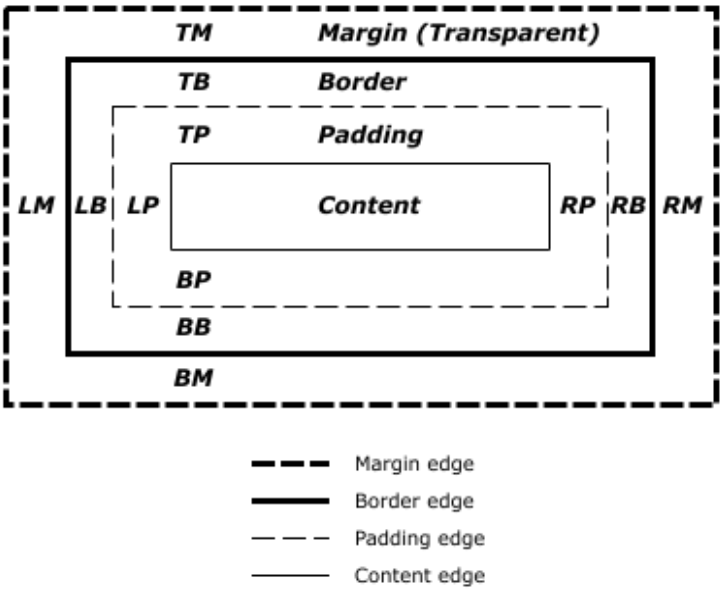


Figure 1: The CSS Box Model. Source: W3C CSS 2.1 Specification.

These style sheets are called “cascading” because of how they handle situations where multiple rule sets apply to a single element. CSS specifies a priority scheme to determine which of the overlapping properties should take precedence. This cascade goes from most to least specific. So if for instance you have a rule set for anchors (“a”) and a rule set for anchors of the class “emphasized” (“a.emphasized”) the properties in “a.emphasized” will be prioritized over the same properties in “a” for anchor elements with the “emphasized” class attribute, as seen in Code Snippet 3.

```
a {
	font : “courier”;
	color : “red”;
}
a.emphasized {
	font : “arial”
	font-style : italic;
}
```

Code Snippet 3: An example of cascading in CSS. Anchor elements with the "emphasized" class attribute will have a different font and style than regular anchors, but will share the color.

This way of cascading is also used when prioritizing which style sheets should take precedence. CSS gives authors three ways of defining style sheet sources in a document: inline, embedded and external. The priority scheme is as follows:

1. **Inline**, inside the HTML document, specified for a single element inside its "style" attribute.
2. **Embedded**, inside the HTML document, defined in its own block inside a "style" element.
3. **External**, a separate CSS file referenced in the document.

CSS also allows using different styles depending on the media type from version 2 onwards. E.g. this can allow for differentiating between regular screen versions and printed versions of a document – giving authors the ability to tailor a document's presentation to different media.

2.1.2.1 Media Queries

Media Queries are defined in the CSS 3 standard as an extension of the old media type detection from CSS 2 [15]. While the old standard allowed for detecting media types such as "screen" and "print", media queries give authors the ability to differentiate between media features as well. Features that can be detected with the new standard include "width", "height" and "orientation". Being able to combine these media types and media features into logical expressions is what constitutes a Media Query. Because of this the new standard contains logical operators to give authors the ability to create these expressions. Operators such as AND, NOT and ONLY can be used to define a Media Query.

```
@media screen and (min-width: 400px) and (max-width: 700px) { … }
@media screen and (min-width: 701px) { … }
```

Code Snippet 4: An example of a Media Query for a screen media type with a screen width between 400 pixels and 700 pixels, and one for screens with a width larger than 700 pixels.

By using Media Queries authors can specify different presentational properties based not only on what type of medium, but the features of any given medium that is viewing the document. While the "screen" type covers everything from a mobile phone to a widescreen TV, specifying the dimensions of screen media, such as max-width, for a given rule set means being able to tailor the presentation more accurately to a specific device, which can be seen in Code Snippet 4.

2.1.2.2 Fluid Grids

A fluid grid is the term given to Web page layouts that are built using relativistic dimensions instead of absolute ones. The crux of this is that elements are given dimensions in the CSS based on percentages and text is sized based on "ems", instead of pixels. The em is a unit of measurement that defines font size relative to the parent element's default. Historically the em unit is named after the capital letter M and 1 em thus takes up the space of M [16]. This means that layouts can flow freely outward and inward as the viewport size changes without breaking, hence the name "fluid grid". The viewport in this context is defined by the W3C as a window or other viewing area on the screen through which users interact with a document [17]. This method of styling layouts is inherently more flexible than the old method of using pixels to define the dimensions of HTML elements. In some sense developers are "rediscovering" this method of creating flexible layouts for the Web, as the Web was originally intended to be a fluid medium, as can be seen from the very first Web page published by Berners-Lee in 1991 [18].

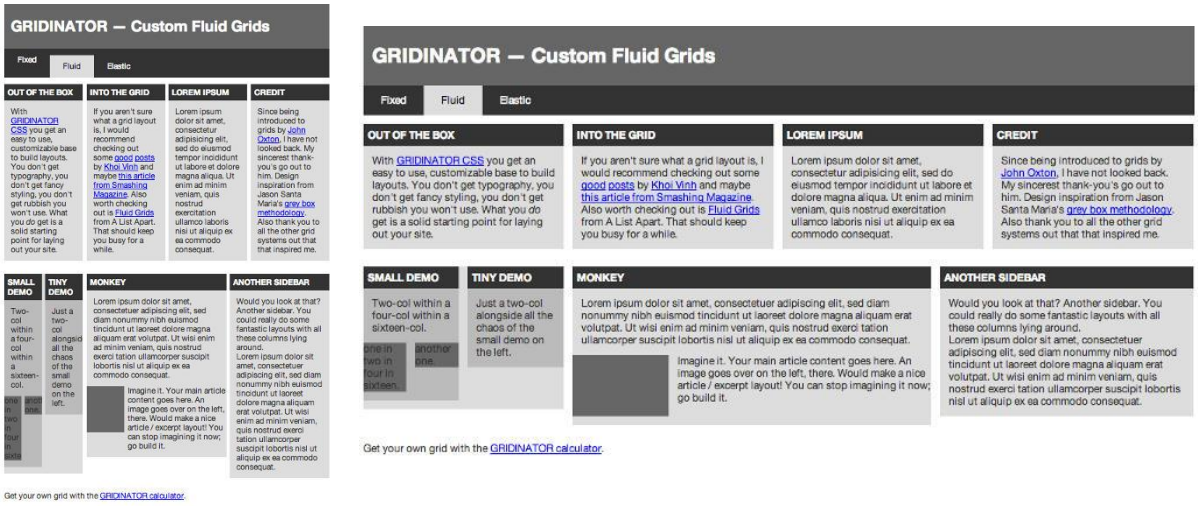


Figure 2: Fluid grid example from Gridinator. Shows the same layout at two different screen dimensions. The boxes can be seen changing their size based on the dimensions of their parent container.

Even though a layout that is built to be a fluid grid will keep its shape based on screen size, it will still break when the difference between the intended screen dimensions and the viewing screen size is big, or the aspect ratio is significantly altered. An example of this is when a Web page designed for widescreen desktops is viewed on a vertical mobile phone screen. This can be seen in Figure 2 where the bottom left boxes are still readable, but single words get broken up on different lines because there is no room on one line. Fluid grids are thus useful for adapting to small changes in viewport size, but not for significantly different devices.

2.1.3 JavaScript (JS)

JavaScript is an interpreted, dynamic, weakly typed scripting language that is commonly implemented in Web browsers to support the creation of dynamic Web pages. It is a multi-paradigm language that supports object-oriented, imperative and functional programming styles. JavaScript's use is primarily client-side, but has gained popularity in server-side applications in later years.

Brendan Eich created JavaScript over the course of 10 days in May 1995 while working at Netscape [19]. The purpose was to create a lightweight, interpreted language to appeal to nonprofessional programmers, similar to Microsoft's Visual Basic language.

```
var displayClosure = function() {
    var count = 0;
    return function () { return ++count; };
}
var increment = displayClosure();
increment(); // returns 1
increment(); // returns 2
increment(); // returns 3
```

Code Snippet 5: A small JavaScript example demonstrating anonymous (lambda) functions and closure

While it was first known as Mocha and later LiveScript, its name changed to JavaScript to coincide with the addition of support for the Java programming language in the Netscape browser. While JavaScript borrows many names and naming conventions from Java, they are otherwise unrelated and have widely different semantics.

Including JavaScript on a Web page can be done in two ways:

- **Embedded**, the code is included in the HTML document in a "script" element.
- **External**, the code is in its own JS file and is referenced in the HTML document.

JavaScript was standardized as ECMAScript in 1997 [19], which is still being developed today. Even though the central part of JavaScript is currently based on the ECMAScript standard, it supports additional features that are not described in the ECMA specification. Many of these additional features are later incorporated into the standard.

Objects in JavaScript can be described in their own notation standard, called JavaScript Object Notation (JSON), though it has its roots in JavaScript, it is language-independent and has parsers available in many languages, for example Java and PHP. It is often used to pass data across networks using protocols such as HTTP.

JavaScript's popularity has only increased throughout the years, and it is now commonplace on nearly all Web pages. This has been especially apparent after the emergence of AJAX (Asynchronous JavaScript and XML), which kicked off "Web 2.0" where dynamic Web pages became the new "thing" to have for both commercial and non-commercial sites alike.

Lately it has become commonplace to write entire Web applications using JavaScript. This has been made possible through the emergence of frameworks that support design patterns previously only used in more traditional programming languages such as Java or Ruby. Along with technologies like AJAX, they have contributed to JavaScript's increasing popularity. Backbone.js and Angular.js are examples of such frameworks. They give authors the ability to structure JavaScript code in an MVC-like (Model-View-Controller) pattern that is commonly used in both regular applications and Web applications today. Being able to structure JavaScript this way means that even large Web applications with a complex code base can be written to be both scalable and maintainable without resorting to using multiple programming languages for various parts of the application.

2.1.3.1 Polyfills and Shims

On top of being used for creating dynamic, asynchronous Web pages, JS is also used to amend compatibility issues in Web browsers. As official Web standards evolve over time, so do the Web browsers who adhere to these standards. This constant evolution causes older browsers to have compatibility issues with new features in specifications created after their release. In a perfect world every user would always use the latest release of their chosen browsers, but for many reasons this is not always a possibility.

Because of this, authors who use features in newly ratified standards will experience that their Web pages do not always appear as intended, or are broken for users with older browsers. To fix these problems, Web developers have looked into the concept of shims. A shim in computing is usually a library between an API and the requestor of said API. A shim intercepts calls to the API and changes either the parameters passed, the operation itself or redirects the request elsewhere. This concept is useful when the behavior of an API changes, e.g. when an author starts using features from a new specification and an older Web browser does not support it. In modern Web development shims are essential to maintain compatibility with older browsers. JavaScript is used in these cases to check for and intercept requests from older browsers. The features are then either emulated by manipulating the DOM with code and other assets, or the Web page is gracefully degraded to not look broken, even though the browser lacks support for the desired features.

Sometimes the concept of shims is expanded to support features that do not even exist in modern browsers. Certain experimental features that are still being discussed by the W3C or do not exist in the HTML working standard can be added to Web pages by the use of shims. These intercepting scripts are commonly known as “polyfills” [20]. While shims have more generalized usage for interception of API’s, polyfills are specifically targeted at compatibility problems in both old and new Web browsers. The main difference between polyfills and shims is that while a shim might have its own API that changes the original in some way, a polyfill works silently with no API of its own. The most common use of polyfills today is to silently fill in support for certain HTML 5 features that old browsers do not support, giving the author the ability to use these

features without writing special control structures in the code to account for supported and unsupported browsers.

2.1.4 Java

Java is a general-purpose, concurrent, class-based, object-oriented programming language with C-like syntax. The main design idea of the language is to minimize the amount of implementation dependencies by allowing developers to "write once, run anywhere" [21]. To achieve this, applications written in Java are compiled to byte code that can run on any Java Virtual Machine (JVM), regardless of computer architecture. Java was developed by James Gosling of Sun Microsystems and released as version 1.0 in 1995 [22]. Even though much of Java's syntax is derived from languages like C and C++, it has fewer low-level facilities than either, though it provides more high-level abstractions and services, such as an automatic garbage collector for memory management.

Java comes in four different editions, with two of them being the most widely used: Standard Edition (SE) and Enterprise Edition (EE). The difference between the two of them is that the EE is meant for building enterprise applications. The EE is built upon the SE and provides a runtime environment and API for developing and running large-scale, multi-tiered, scalable and secure network applications. Java EE is in other words an API that builds on Java SE to make it more network- and Web-oriented for enterprise use.

Java is, as of 2013, one of the most popular programming languages in use according to the TIOBE Programming Community Index [23], particularly within client-server Web applications. It provides developers with a mechanism called Servlets, which offer services for extending the functionality of Web servers and accessing business systems on the server. Servlets typically generate HTTP responses from HTTP requests from the client. A typical response is an HTML document, as shown in Code Snippet 6.

```
import java.io.*;
import javax.servlet.*;
public class Hello extends GenericServlet {
	public void service(final ServletRequest request, final ServletResponse response)
		throws ServletException, IOException {
		response.setContentType("text/html");
		final PrintWriter pw = response.getWriter();
		try {
			pw.println("Hello, world!");
		} finally {
			pw.close();
		}
	}
}
```

Code Snippet 6: A Java Servlet example. The service method overrides the one from the GenericServlet class. It attaches a HTML document that is sent with the HTTP response.

2.1.4.1 Spring Framework

Spring is an open-source application framework and Inversion of Control (IoC) container for Java. Rod Johnson wrote it in conjunction with the publication of his book "Expert One-on-One J2EE Design and Development" in October 2002 [24], and it was first released under the Apache 2.0 license in June 2003. The main idea behind Spring is to address the complexity of enterprise application development in Java.

Spring contains several frameworks that make up the whole system. These include an IoC container, Aspect Oriented Programming (AOP) framework, Data Access framework and a Model-View-Controller (MVC) framework, among others. They all help by providing abstractions and services that assist in managing large applications.

The IoC container of Spring is what allows applications built with the framework to keep objects loosely coupled through dependency injection. It is a design pattern where objects are instantiated at runtime by the container, having the dependencies of which objects need which services defined in a configuration file. The container instantiates all needed objects and wires them together by setting the necessary properties and determines which methods will be invoked. Dependencies are either assigned through properties, e.g. setter methods, or through constructor arguments.

2.2 Concepts

2.2.1 Progressive Enhancement and Graceful Degradation

The “Web gurus”, tech writers and user experience professionals Steven Champeon and Nick Finck first suggested the concept of Progressive Enhancement in 2003 in response to the perceived necessity of mixing structure and presentation in HTML documents at the time [25]. This necessity came from the browsers containing several bugs when it came to adhering to the CSS specification, among other things. Web developers were forced to resort to hacks and “Graceful Degradation” in their Web documents to support legacy browsers and modern browsers with buggy CSS parsers. Graceful Degradation, in a nutshell, means creating Web documents with modern technologies that are based on current standards, but having the document still be readable in older browsers and browsers that do not support these technologies. This is achieved through detecting this lack of support and hiding the unsupported parts of the document from being viewed in these browsers. The problem with this, Champeon argues, is that it wrongly assumes that browsers always evolve to be better, faster and more powerful. The idea is that creating Web documents focusing on presentation simply adds unnecessary weight to them without adding anything to the message to anyone but users of modern graphical browsers. He further comments that emergent devices (at that time) such as mobile phones and PDAs with browsers simply do not have the capacity to show pages with this extra presentational data – which is even more relevant today considering the widespread use of browsing the Internet using modern mobile devices.

Progressive Enhancement attacks the problem of supporting older or less powerful browsers by having them be the baseline target of the document, i.e. pure HTML markup with minimal CSS styling, focusing on the content and not the presentation. As more advanced capabilities and features are detected in browsers, the presentation of the document can be enhanced by using more elaborate CSS, scripts and media to add on to the already existing baseline. Champeon argues that this reversal will help support the separation of presentation and markup by focusing on the content first, as well as increasing accessibility of Web pages and making it easier to tailor Web sites to be sensitive to client capabilities.

2.2.2 Web Content Management Systems (WCMS)

A WCMS (commonly shortened to CMS – Content Management System) is a software system, most commonly a Web application, which allows publishing, editing and modification of Web content from a central interface. The most common function of a CMS is to present Web sites and support Web site authoring, collaboration and administration for users with little to no knowledge of Web programming languages or markup languages. Users can create and modify content like text, images and other media and the CMS will put it all together in a chosen template and publish it as a fully fledged HTML document when the user chooses.

CMSs are massively popular for building Web sites today, as they require a minimal amount of technical knowledge to create and publish user-generated content. CMSs such as WordPress, Joomla! and Drupal make website maintenance possible for people with limited expertise, without the intervention of a Webmaster. They also provide the ability to allow users to contribute content to a site without having to think about the technical aspects of Web publishing. This ability to easily author and publish content on a Web site has helped in the proliferation of the dynamic, content-driven personal sites, such as blogs, that have largely replaced the static personal Web sites of old.

While these systems usually can be set up easily by non-technicians “out of the box”, they usually support being expanded by user-developed modules, often called “plugins”, that expand on or create brand new features for a Web site. These plugins are commonly written in the CMS’s programming language (Java, PHP etc.) and their creation thus requires knowledge of said languages.

2.2.3 Responsive Web Design (RWD)

Making Web pages respond to changes in its environment, such as screen size and orientation, has become increasingly important now that mobile devices are taking over as people’s main way to browse the Internet [1]. Previously, building a separate site for mobile devices and redirecting to it as the server sees fit has been used to achieve this. These solutions give the developers multiple code bases to maintain instead of just one. They are also static in that they are built for desktops and, most commonly, phones. So

which one should be served to a tablet, or a TV? Should developers be tasked to create a new, separate version of a Web site every time a new device with a different form factor comes along? This would be extremely cumbersome and could increase the complexity of any Web development project to unsustainable levels.

RWD is a method that suggests keeping only one code base that has its layout changed as the environment changes. I.e. having the Web page's CSS alter how the page looks based on certain device class definitions, as well as switching from static to fluid layouts that are relative to screen size, instead of fixed width. This will allow the Web page to adjust to minor variations in screen size, and not just massive changes such as going from a widescreen desktop monitor to a mobile phone. The "Web guru", author and speaker Ethan Marcotte first suggested it on A List Apart in 2010 [26], and later elaborated on the idea in the book "Responsive Web Design" [2], which was published in 2011.

RWD in itself is largely based on fluid grids, flexible media and media queries. Flexible media refers to having media such as images and video change size (flex) along with the viewport. The idea is to use these media queries to alter the CSS used for elements on a Web page based on the device's screen size, and let the fluid grid fill in the smaller gaps between devices of the same class. Often the designer predetermines at which resolutions the layout of the page will use different CSS defined within the media queries. These changes in layout are done on the client side, since it is all determined in the Web page's CSS. While the layout is changed and elements may be hidden from users on mobile devices by using media queries, all the content defined in the page's markup is still downloaded by the browser. This is an aspect of RWD that has been criticized, because it leads to a lot of superfluous data being transferred to devices with limited resources [3]. Another commonly observed problem with RWD is that media queries are not always supported by browsers, both mobile and otherwise, especially older ones. This is considered a problem because unsupported browsers will end up having trouble parsing the CSS of a page using media queries. On certain browsers, such as Internet Explorer 7 and 8, this can be fixed by using polyfills written in JavaScript, but this is not possible in older mobile browsers that do not have sufficient JavaScript support.

2.2.4 Mobile First

Many of the aforementioned issues with RWD arise from the fact that developers still use the desktop version of the site as the baseline for the development process. This is understandable considering how the desktop computer has been the main way of accessing the Web for more than twenty years. Making the desktop experience gracefully degrade for mobile devices has been the way to go. As mentioned in the RWD section, this can lead to Web pages inheriting elements from the desktop version that might not even be visible because of media queries, yet they are still downloaded.

The book "Mobile First", written by the internationally recognized user experience designer, author, speaker and previous Chief Design Architect at Yahoo!, Luke Wroblewski suggests that attacking the problem from the opposite direction might be the way to go [27]. Developing the site for mobile devices first will lead to the desktop benefitting through a more focused design centered on relevant content, he claims. The Mobile First mindset allows developers to embrace the constraints of mobile devices to create services that only deliver what the customer actually wants, cutting out everything that is not totally necessary. Focusing on the constraints means looking more closely at the performance of a given page to make it function properly even in conditions of low bandwidth and weak computing power. It also means looking at the capabilities of mobile devices and using the platforms to their full potential, using what might not be thought about if the basis of development is a desktop computer. The concept furthers the idea that Web pages or Web applications can be made richer by adding context awareness through a mobile device's sensors and inbuilt services such as GPS, compass, gyroscope, as well as telephone and camera.

Making the desktop site from a mobile baseline means progressively enhancing it as the capabilities and screen size of the device improves. This means doing the polar opposite of Graceful Degradation. The idea is that this way of thinking will sort a lot of performance issues itself simply because the designs and the use of scripts and media will be more limited due to lack of screen space and resources on a mobile device. Wroblewski further argues that this approach also helps with regards to making images and other media more bandwidth-friendly; that it will make browsers download mobile-

optimized media first – only downloading high-definition desktop-optimized media if the device can handle it, or the user wants it.

2.2.5 Device Detection

Device Detection in the context of the Web refers to software that identifies the type of device that is making requests to a Web server and either redirects to a Web site made specifically for that device class or adapts the Web page to a format that fits the device class.

2.2.5.1 Device Description Repositories (DDR)

The Device Description Working Group (DDWG) of the W3C proposed DDRs in 2006 [28]. They are supposed to contain information about Web-enabled devices that is made available through a standard API. These repositories contain a core vocabulary of device properties such as screen dimensions, input mechanisms, supported colors etc. Several implementations exist, both of the open and commercial kind. Examples include WURFL, DeviceAtlas and OpenDDR. These kinds of systems have existed since before the DDWG started work on a standard, though. WURFL, for instance, was released in 2001 [29]. The standard DDR API was published as a W3C recommendation in 2008 [30].

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/537.31 (KHTML, like
Gecko) Chrome/26.0.1410.65 Safari/537.31
```

Code Snippet 7: An example of a UA string. This represents version 26 of the Google Chrome Web browser.

DDRs are present on the server and work by parsing the UA string, an example of which can be seen in Code Snippet 7, from a requesting client and matching it as best as possible to data found in the repository. When the best match is located in the repository a set of features supported and the type of device is returned through the API. These repositories need to be maintained by people and thus have no way of accurately interpreting a UA string that is completely different from anything already stored in the database. These features imply that DDR's are inherently not Future Friendly, as they cannot work autonomously because their database has to be manually updated.

2.2.5.2 Client-Side feature detection

While the most common form of device detection is done server side, as mentioned earlier, it is possible to do this kind of detection more accurately client side. In this case a test suite is sent to the client in the form of JavaScript and is run to check for what features are available in the requesting browser. The data gathered from this can be used directly through CSS to adapt the page according to the browsers capabilities, somewhat similar to Media Queries, but with more features being available to be queried. The information can also be sent back to the server, either through an AJAX call, or by storing the information in a cookie and reloading the document.

A popular system for client-side feature detection is Modernizr. Modernizr is primarily used to detect HTML 5 and CSS 3 features in browsers. The motivation behind this is that, even though many people still use browsers without decent support for these standards, authors should be able to utilize the latest features in these standards without having to worry too much about compatibility issues. Modernizr thus offers a way to detect these features so that authors can easily detect compatibility issues and either provide fallbacks or have polyfills transparently supply the functionality that is missing through what they call “conditional resource loading”. The system works by having its detection script add classes to the HTML-element of the document that describe what features are supported. This information can then be used to adapt the page through class-selectors in the CSS and by having the conditional resource loading provide the polyfills needed.

2.2.6 Responsive Design + Server Side Components (RESS)

RESS is a concept also suggested by the previously mentioned author of “Mobile First”, Luke Wroblewski [7]. It combines the flexibility of RWD with the capability of having the server decide what markup to serve the client. The idea is to have a single template that defines the layout of the page, but have selected components adapted with different implementations on the server. This way we get the flexibility of RWD with the performance issues fixed by tailoring the markup on the server. This leaves the client to adapt the layout like with RWD, but without having to download unnecessary media and scripts. The server can optimize things like source order, URL structure, media and

application design before sending any content to the browser, all without relying on user agent (UA) redirects to entirely separate device class specific code templates. Wroblewski argues that the problem with these standalone full-page code templates is that they are almost guaranteed to contain duplicate code that also exists in the implementations for other device classes. Being able to reduce or completely eliminate duplicate code will make life easier for developers as well as increasing the maintainability of the system.

Using this method we can have a server-side solution without relying on redirects. The same URL can be kept for each implementation since the server generates the whole adapted page, just like with RWD. Keeping the same URL will make it easier for users to navigate a Web site, as they no longer have to think about using different URLs for different versions. Fluid grids and media queries will still handle the layout, but the components themselves will be optimized according to the capabilities and features of the device sending the request. Known problems with RWD such as flexible images can be solved using this method as image components on a page can have their markup changed to reference mobile-friendly images if the server detects such a device.

There is one problem with having detection done on the server, though: it relies on being able to accurately detect what kind of UA is making the request. The reliability of server side UA detection has been heavily debated [31], and several solutions exist for accurately interpreting UA strings on the server, such as the aforementioned DDRs. The problem is that UA strings are not always reliable: they can be spoofed through proxies and some of them do not contain enough information for the server to know what kind of UA is making the request. The solution of redirecting based on these strings falls flat as soon as someone fakes the UA string or otherwise blocks the server from making what amounts to an educated guess. Being reliant on the UA string alone means that UAs that do not send a meaningful string will be subject to receiving a default “fallback” page for unknown UAs.

The advantage of RESS over full-page redirects and pure RWD, Wroblewski claims, is that a single set of page templates defines an entire Web site for all devices, but specific components within the templates have device-class specific implementations that are

rendered server side. This relieves the burden on the client to handle the whole layout and presentation as well as allowing the author to control the URL structure, media and application design [7].

2.3 Related work

2.3.1 RESS with Detector

Detector is a flexible server side UA feature detection system created by Dave Olsen, a programmer and project manager at West Virginia University, and written about in his own blog [10]. The idea behind it is to never assume you know everything based on just a UA string, and that you should never run feature tests unless you do not know which UA you are running in. Based on this he skips the predefined DDR altogether and makes the system learn the features each new UA it encounters through a client-side test suite.

The system is split into two parts: the client-side test suite, based on Modernizr, and the server-side detection logic that stores and looks up UA strings and decides what to do with them. The client side part is a Modernizr test page that is sent to the client if the UA string has not been encountered, or if certain tests have been set to run on a per-session or per-request basis. The results from these tests are saved in a cookie and stored on the server upon a page reload. The server side logic is responsible for storing the information gathered by the client-side tests as well as gathering useful information from the UA string itself, such as operating system and device name. This method of user agent detection (or more accurately: feature detection) not only removes the need for a central DDR, it makes the system Future Friendly by allowing it to figure out an unknown device's features from the device itself, instead of a database that is solely relying on the UA string for information. Of course, this means that a new device that has not been encountered by the system will be subject to a barrage of tests that will slow down the loading time of a page and take up valuable resources. But it will only happen once: the very first time this new UA string is detected. Each subsequent request from the same UA will get served the appropriate markup straight from the server, without having to run the tests on the client.

Deciding which markup to serve a given UA is done by the implementation of Detector on a given server. Detector gives the ability to define device "families" that decide what features uniquely identifies a class of device you want to tailor the page to, for example "mobile", "tablet" and "desktop". Families are defined in a JSON file that is loaded by Detector and used to look up which family is best suited to the requesting UA. Attributes for a family can take two forms: Boolean attributes gotten from the ua-parser.php and a "features array" that can contain any features detected by the Modernizr client side tests. All attributes and features in the feature array have to be evaluated to "true" for a family to be selected. The JSON file containing the family definitions are read sequentially, and as such have the most general case defined last and most specific case first to find the best fitting family for any given UA. These device families are central to creating a RESS solution by using Detector, as they are what we use to differentiate between different device classes.

To build a RESS solution by using Detector there is one extra thing needed: a template framework, such as Mustache, Handlebars or Jade. Without some form of template system RESS is not possible, as you would have to serve an entire page defined by the device class, instead of tailoring specific components while keeping the base markup common between all classes. Which template system is used is not as important as arranging the templates into a well structures file-hierarchy. The reason for this is to allow for easy switching between template-partials after having Detector decide which family a UA belongs to. Following Luke Wroblewski's example of having a page with different navigation for different devices [7], we could create a Web page using Mustache for templates with the given file-hierarchy for partials, as seen in Code Snippet 8 and Figure 3. Each Mustache file in this example will contain HTML that is tailored to the device class it belongs to.

```
[...HTML, head tags etc. ...]
<body>
      {>header}
      [...some content...]
      {>footer}
</body>
[...HTML closing tags...]
```

Code Snippet 8: HTML Markup with Mustache partials for the header and footer of the document.

```
\base
      header.mustache
      footer.mustache
\mobile
      header.mustache
      footer.mustache
\tablet
      header.mustache
\desktop
\tv
```

Figure 3: File hierarchy for Mustache partials in the Detector system.

The markup has the header and footer set as Mustache partials. This means that whenever the page is loaded the partials will be filled by whatever markup is in the templates in the folder of the family chosen by Detector. In the base case we have preset definitions of both the header and footer sections. For the mobile family of devices we have overwritten both the header and footer – perhaps to move a lot of navigation from the header to the footer for more comfortable one-handed use. In the case of the tablet the header will be overwritten while the footer is left untouched. In the case of both desktop and TV no changes are made as the templates in the base folder are meant for use with larger screens. This way of organizing the templates lets the system look in the folder of the chosen device family first, while filling in the remaining partials with what is in the base folder – removing a lot of code that would otherwise be repeated throughout each family-specific markup. Falling back on a base directory is not supported in Mustache by default, but Olsen has made the small tweaks necessary for this functionality in his own fork of Mustache's loader [32]. While this kind of structure is invisible to the user, it makes for a lot more readable code, and keeps the code base maintainable even if it grows in size, by virtue of its modular design. Changing the order and layout of different components is just a matter of changing the base HTML file, leaving the device-specific code untouched. It also allows for adding family-specific components later by simply adding new partials in a specific folder. E.g. if the base navigation in the header does not utilize the screen space of a 50" TV well enough; it is just a matter of adding a *header.mustache* file to the TV folder and it will replace the base markup with TV-tailored navigation whenever Detector gets a request from something it detects as a TV-family UA.

Not only presentational elements can be turned into components using the RESS with Detector. Includes such as scripts and CSS markup can be put into partials that are specific to each device family. Doing this will ensure that devices only ever download files and markup that is useful and visible to the user; skipping all the fluff that is present in most RWD solutions and solving the problem with using "display:none" as the only way of hiding elements from users on unsupported devices. This way the system supports both "progressive enhancement" and "graceful degradation" of a Web page without much extra work, as the system will detect which partials – and thus scripts, CSS, and media – to download for any given device.

Since the family system is completely independent of the base markup of a page, it opens for increased adaptability from the developer's standpoint. If a certain mobile device falls outside of the scope of the normal "mobile" family definition, a new family can be created to cover this. While modern "smart phone" browsers today support almost as much JavaScript and CSS as a desktop computer, many older phones do not, and many of these are still prevalent in many parts of the world. These "feature phones" have browsers, but they often lack much CSS support and rarely support JavaScript to any useful degree. Assuming these phones fall outside of the "mobile" definition used above, the server will fall back to the default components, which are designed with desktop in mind. This is definitely not desirable on these types of devices. Creating a family to cover them is needed, and can be done by creating a family called, for example, "mobile-basic" that covers the features present on such devices. This requires nothing more than adding this new definition to the family JSON file and tailoring the necessary partials for these types of devices, the rest of the code base does not need to be touched.

Detector offers a fairly straightforward solution to the problem of making the server more involved in RWD. With a tweak to make the template system able to "failover" to a base directory in the case of missing partials is all that is needed to create a RESS system. It is also flexible, with its family system allowing for any kind of template system to be used, which gives developers increased freedom in building their Web pages. The family system also separates the logic of detecting which partials to use from the presentational markup in the HTML and template files, keeping the code base clean and maintainable while being Future Friendly. While it does not solve the problem of "dumb

content" (which should fall to the CMS) and context aware coding (which should be done by the editor) as mentioned by Sæterås [6], it does go a long way to create a possible solution for the server side of things. Detector is currently in production on the West Virginia University home page [33].

2.4 Summary

In this chapter we have looked at the technologies, languages and concepts underlying the thesis: HTML, CSS, JavaScript, Java, Device- and Feature Detection, Responsive Web Design, Mobile First and RESS. We have also looked into related work in the same field, focusing on the Detector project by Dave Olsen. The goal of this has been to make the motivations behind the implementation of my system clearer, giving an understanding of the technologies it is based upon, and putting the thesis in a historical context. To be able to move forward to create solutions for today's problems, it is important to understand what has already been done and what kind of problems existed before.

3 Enonic CMS

Here we will look at the parts of the Enonic CMS that are relevant for the development of my server-side feature detection system, how they work, how they interoperate and how they support the development of plugins for the CMS.

3.1 Background

Enonic CMS is a Web content management system (WCMS) created by the Norwegian company Enonic AS and was first launched as version 1.0 in 2001. It is based on Java Enterprise Edition (JEE) and utilizes many open-source technologies such as Spring for inversion of control, Hibernate for object-relational mapping database abstraction, and Saxon for XML and XSLT processing.

Enonic is meant to function as a software platform for medium to large organizations [34] and as such provides the tools for development and publishing needed to do this. As with many modern CMSs, it is built to support content creation and publishing by users and not just developers. To do this it has a Web-based portal that gives users a user-friendly way of creating, managing and publishing content. It also supports "in context editing" (ICE) of Web pages – the ability to edit the content of a page while viewing the page itself. The system is widely used in the Norwegian public sector [35], which is one of the reasons for choosing it for the implementation of the system.

Because the system is based on open-source technologies, it aims to be platform-independent. It supports all common operating systems, servlet engines and relational database servers. The Enterprise Edition also supports directory servers such as the Lightweight Directory Access Protocol (LDAP) and Microsoft Active Directory (AD) for handling enterprise-level directory information.

Enonic comes in two different editions: Enterprise Edition (EE) and Community Edition (CE) [36], where the latter is open-source and free to use under the Afferno General Public License 3.0 (AGPL 3.0). They are mostly similar, but with the EE supporting more

enterprise-oriented elements such as directory servers, load balancing and dedicated support. The EE is available under the Enonic Commercial License.

3.2 Datasources

A datasource in Enonic CMS is a collection of one or more Java method calls used to retrieve data from a database or other sources [37]. Methods invoked in datasources return XML or primitive types, and only accept primitive types as arguments. To supplement the native library of methods available in datasources, new ones can be added through plugins. Every call to a datasource method uses an instantiation cache that stores the data gotten from the initial call, so that subsequent calls to the same method within the page will not trigger an actual call, but rather get the return value or XML from the cache.

Datasources are defined as XML and can contain several method calls. Each call contains a name attribute for the method and a list of parameters that specify the arguments to be passed to the method.

```
<datasources>
      <datasource name="..." [condition=" "] [result-element=" "] [cache="false"]>
            <parameter name="...">value</parameter>
            ..
      </datasource>
      ..
</datasources>
```

Code Snippet 9: An Enonic CMS datasource. Attributes in brackets are optional.

The condition attribute may contain a condition expression. The value is commonly a Java Unified Expression Language (Java.el) expression that dictates when the method call should be executed. The result-element attribute specifies the name of the root element for the result-set XML that is returned from the method call. The cache attribute states whether or not the result set should be stores in the instantiation cache for subsequent method invocations.

3.3 Device Detection and Classification

Enonic CMS supports device detection on the server [38]. It does this by allowing an XSL-based device-classification script to be referenced in the site properties of the CMS. The device-classification script gets passed data from the CMS in the form of an XML containing values from the HTTP request, and the user. This is similar to having device detection with Device Detection Repositories, as the most common resource that can be used for device classification is the UA string that is present in the HTTP request header. No information about the supported features on the UA, apart from the UA string, is passed to the script in the native XML.

The output of the XSL script is a string describing the detected device class based on the data from the XML. Detecting a device class is done by a conditional block in the script that matches data from the XML against user-defined regular expressions. Whichever regular expression result in a match decides which device class is passed to the CMS. The output is attached to a context-element in the datasource result XML of all pages and portlets in the CMS, and is thus available for tailoring the site to the detected device class.

3.4 Plugins

Enonic supports development of plugins for extending its functionality [39], this is done using Java and Spring. Plugins are packaged in OSGi bundles, which are normal JAR files with extra metadata called the “Manifest” that allows for the modularization that is needed for plugins in the Java system of Enonic CMS. Maven is used for building plugins. It handles all the dependencies for the plugin as well as packaging the JAR file for deployment. Enonic has also created a Maven plugin that simplifies the process of packaging the plugin into an OSGi bundle that is compatible with the CMS. To deploy a plugin the JAR is moved into the plugin directory under the Enonic installation directory, or wherever the plugin directory has been defined in the CMS’s configuration.

Developing a plugin for Enonic is essentially creating a set of extensions packaged into a JAR file. In Enonic CMS this means extending Java classes that are part of the Plugin Environment API, as can be seen in Figure 4.

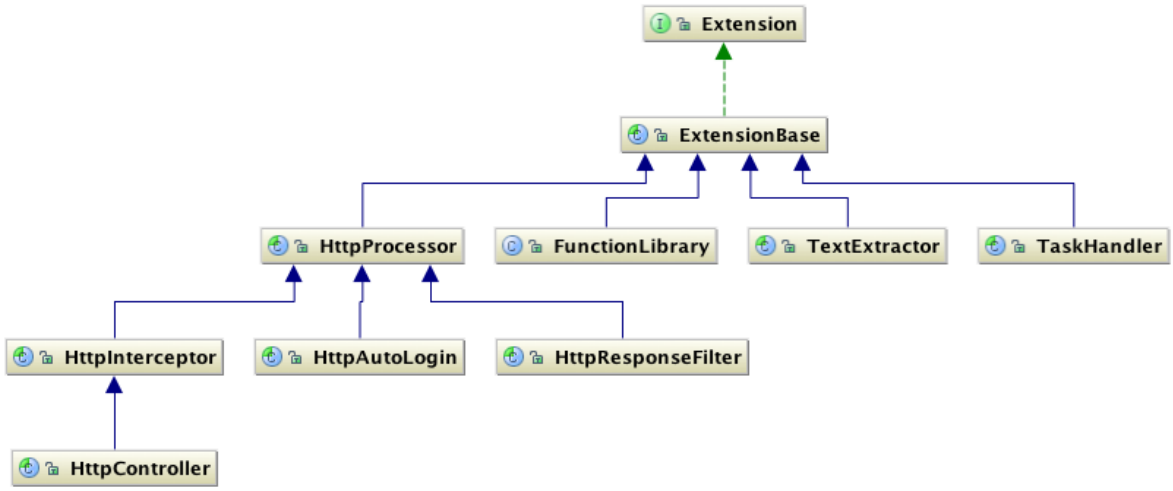


Figure 4: The Enonic Plugin Environment API. Source: Enonic AS

Most of the classes in the API can be extended, with two exceptions:

- **Extension** is an interface and cannot be extended, but is implemented by **ExtensionBase**; **ExtensionBase** and **HttpProcessor** are super classes that should not be extended directly.
- **FunctionLibrary** should not be extended directly, but should be used as a Spring bean class to define a **FunctionLibrary** extension. All public methods in the extended class can then be invoked from datasources in the CMS.

The classes that extend **HttpProcessor** are mainly meant for handling HTTP requests and responses in some way, such as filtering based on the type of request the server receives, automatically logging in a user based on data in the request header, or taking full control of the HTTP response.

The **TextExtractor** class is meant to extract text from various binary document types, such as PDF, MS Word and RTF, to index it for the CMS's search engine. It can be extended to support text extraction from formats that are not natively supported by the CMS.

The **TaskHandler** class handles scheduling of code execution, via extensions. Scheduling when execution should occur is done in the Spring bean of the extending

class using a special property called “cron”. Cron is a reference to the Unix task scheduler of the same name. The Enterprise Edition of Enonic also supports running tasks scheduled through the **TaskHandler** in a cluster.

The **FunctionLibrary** class is meant to allow for extending datasources in Enonic. They give developers the ability to create an API that gives users access to database information from the CMS itself. These functions must only take parameters of primitive types, i.e. integers, Booleans etc., and can only return primitive types or XML. As previously mentioned a **FunctionLibrary** extension cannot extend the class directly, but should be defined as an extension in the **FunctionLibrary** Spring bean. This is because of how Enonic handles this class and gives access to function invocation from its datasources.

3.5 Summary

We have looked at the technologies and solutions that underlie the Enonic CMS, how it supports the development of plugins and how its plugin API is structured. The point of this has been to provide an understanding of what lies beneath the solution I will present in the following chapter.

4 Implementation

In this chapter we look at the implementation of the server-side feature detection plugin for the Enonic CMS. I will present how I approached the conceptualization of the system, as well as how it was actually implemented. I demonstrate the technical considerations that were made and how the chosen technologies were used to create a functioning server-side feature detection system for the CMS, inspired by the ideas of Dave Olsen and his Detector system, which was presented in chapter 2.

Enonic has a built-in system for doing device detection, as mentioned previously. Though the system only detects the very basic data that can be extracted from the UA string. Because it can be a potential improvement over the built-in system, we will look at the development process of the server-side feature detection plugin for Enonic CMS, from conceptualization to implementation.

Later we will base our discussion on this implementation and the subsequent tests of its performance to establish the viability of using the system in the back end of an actual Web site, considering both the concepts it is based on and its technical merits.

4.1 Conceptualization

Looking at the plugin support for Enonic it is evident that a server-side feature detection system is feasible to implement as a plugin for the CMS. The native device classification system is also lacking in the amount of data it makes available to the user, providing further incentives to improve upon it. As mentioned in chapter 2, being able to identify and catalogue UA-specific features on the client and store it on the server can help developers and users tailor their Web pages to specific device families more accurately. We thus posit that a server-side feature detection plugin can be a valuable addition to Enonic.

When considering the plugin environment API, two different approaches to implementing a server-side Feature detection system were identified. They would both

extend **HttpInterceptor** to intercept requests and detect the requestor's UA features on both the client and server side, and then either:

1. Store and/or retrieve the data from a separate database and forward the request with the data attached to it, so it is available in the CMS's device class resolver script.
2. Store the data if the device's features are not already in the separate database, and then use an extension of **FunctionLibrary** to make the device features available from Enonic's datasources.

The first solution would be the most optimal; as it would not require users to change the way they handle device classes on their Web pages. This would make the plugin more practical to use and integrate into already existing Web sites built in Enonic. While this solution was conceptually sound, it was not necessarily feasible. This is because the way Enonic builds its device class resolver XML data was undocumented and not obvious. It was also not apparent if it was possible to attach any data to the request in an **HttpInterceptor** extension, and whether it would be sent to the device class resolver script.

I figured out from early on that the second solution should work, as making data from a database available through a datasource is what **FunctionLibrary** extensions are meant to do. Though it would require some extra complexity in the plugin, as well as in its use. It would need an extra extension and an alternative to the XSL-driven device class resolver script system that is built into Enonic. This approach would need a device-family classification system that would make the plugin more akin to Dave Olsen's Detector system.

4.1.1 Storing the gathered data

Storing the data gathered by the plugin should be done in its own database. The reason for this is that the device feature data gathered is not a part of the native system and should not be stored in the same database as the contents of the Web site. A low maintenance, lightweight database system should be used to minimize the amount of

overhead caused by adding another persistence system to the CMS, as well as simplifying the setup and configuration of the plugin for users. This solution will probably function as a better alternative to storing the data as a content-type through the Enonic Java API. This is because storing and retrieving data directly from a database will normally be orders of magnitude faster than going through the whole call-stack that is invoked when retrieving content through the Enonic Java API.

For a simple database some form of NoSQL system, either document-based such as MongoDB or CouchDB, or a key-value store such as Redis or Voldemort could be useful. A relational database could also be used, but it would impose an overhead both in terms of setting up and maintaining the system, by nature of its strict structure. The plugin should nonetheless be constructed to be database-agnostic to allow for connecting to any arbitrary database system.

4.2 Implementation

For the implementation it was decided to attempt both approaches to see if they both worked, and potentially which one was the most efficient in terms of speed and ease of use.

4.2.1 Technologies

Modernizr was chosen for the client-side test suite, as mentioned in chapter 2, with all available tests, licensed under the MIT License. For the server-side UA string parsing a version of UA Parser created by Twitter, Inc. was used, licensed under the Apache License 2.0. MongoDB, a document-based NoSQL database system available under the GNU Afferno General Public License 3.0, was chosen for the database.

The choice of using Modernizr for client-side tests and UA Parser for UA string parsing was made firstly because they are both used in Dave Olsen's Detector system [32]. Secondly because Modernizr is the industry-leading feature detection system, and UA Parser is very lightweight while retaining all the functionality needed to gather the necessary information from UA strings.

MongoDB was chosen because it is lightweight, easy to set up and removes all object-relational-mapping work and schema planning needed when using relational databases. As previously mentioned any database system can be used, but document-databases such as MongoDB suit the use-case of the plugin better than relational databases. This is especially true when considering the relatively small amount of data that needs to be stored. The amount of data and the complexity of the data structure do not warrant spending much time planning out a relational schema for it. The data that needs to be stored is essentially a single object containing key-value pairs in which the values are either objects themselves or Booleans. JSON, which is the format MongoDB stores its documents, is well suited for this kind of use case. Because Modernizr is a JavaScript tool its test results are stored in a JavaScript object, which is, as mentioned in chapter 2, represented with JSON. This makes the translation from the Modernizr test result object to a MongoDB document simple. MongoDB also has a flexible schema that allows the document structure to be changed and expanded without invalidating or corrupting older documents [40]. This can prove quite useful when considering the future friendliness of the plugin; allowing it to have the set of detectable features from Modernizr expanded without having to edit any database schemas.

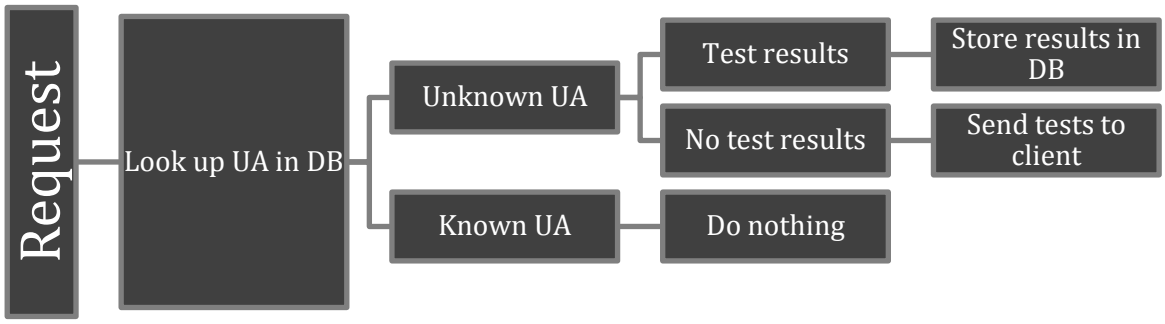
The plugin itself was by necessity written in Java using Spring, and Maven for dependency handling, building and deployment. It needed two extensions of the Enonic Plugin Environment API that we looked at in Chapter 3: **HttpInterceptor** and **FunctionLibrary**. The latter was only necessary for the second approach, as noted under the Conceptualization section.

4.2.2 Application flow

The **HttpInterceptor** extension is what intercepts and handles the HTTP request that comes from the client before it is passed to the CMS. It contains two methods that must be overridden: **preHandle** and **postHandle**. As their names suggest they are invoked before and after the CMS has handled the request, and they accept an **HttpServletRequest** object and an **HttpServletResponse** object as arguments. These objects are passed to them by the CMS servlet that receives the request. All of the business logic must thus be rooted in the **preHandle** method, as it is necessary to

intercept the request before any HTML is served by the CMS, i.e. before the HTTP response has been generated.

It was quickly surmised the first approach mentioned in the Conceptualization section would not be feasible, as the **HttpInterceptor** lacked the ability to pass any additional information up the request chain to the CMS. This left the second approach of having a **FunctionLibrary** extension that accessed the database to retrieve and process the data stored by the interceptor extension. The program flow in the overridden **preHandle** method is thus as follows:



1. Get the UA string from the header of the request and look it up in the database.
2. If the UA string is present in the database, go to 3, else go to 4.
3. Do nothing – the data will be fetched by a method invoked from a datasource later, pass the request up the chain by returning true. End.
4. Check if a cookie with the correct ID is present in the request, or if the GET variable indicating the lack of JS support on the UA set, indicating that client-side tests have already been run on the UA.
    a. If a cookie is present, parse the test results from the cookie and store them in a database object, go to 6.
    b. If the no JS support variable is set, store the nojs indicator and go to 6.
    c. If not present, generate the correct HTML markup and JavaScript code to send Modernizr tests to the client.

i. Generate HTML markup.
ii. Add Modernizr code and cookie-generator code to the markup.
iii. Add “noscript” element to the markup to catch UA’s lacking JS support.
iv. Send the generated markup to the client and return false to stop the request from going further up the request chain. Go to 5.

5. On the client the page will look blank to the user, but will only be visible for the time it takes the JS code to execute:
    a. If the browser supports JS Modernizr will run its tests, a cookie will be generated and the page will be reloaded using JS. Go to 4.
    b. If the browser does not support JS a “noscript” element in the generated markup will add a GET variable to the end of the requesting URL before reloading the page, indicating the lacking JS support to the server. Go to 4.
6. Get information from the UA string.
    a. Parse the UA string using UA Parser.
    b. Store the collected data in a database object and go to 7.
7. Put both the client-side data and server-side data into a common database object and store it in the database. Go to 3.

The **postHandle** method does not need to do anything, as the goal of the **HttpInterceptor** extension is to intercept the request and get the necessary data from it, not to manipulate the response on the way out. Manipulating the markup sent in the response should be done by the CMS based on the data and family definitions gotten from the plugin’s **FunctionLibrary** extension. Changing the response directly through an **HttpInterceptor** extension would not adhere to the RESS principles presented in chapter 2.

4.2.3 Plugin configuration

Enonic supports configuration of plugins using property files. These files define key-value pairs that can be referenced within the plugin. A default property file is present in the plugin’s own JAR file, with the possibility of having external property files overwriting the default values. The plugin has several values set in the default property

file to give users the ability to configure their database and reference external files such as their own customized Modernizr JavaScript and device family definition JSON.

```
mongodb.host = localhost
mongodb.port = 27017
mongodb.dbname = mongodetector
mongodb.collection = useragents

modernizr.uri = modernizr-2.6.2.min.js

families.uri = families.json
```

Code Snippet 10: The default.properties file

An external property file can be used to define a custom hostname or port for the MongoDB instance being run on any server. It can also define the URI for the Modernizr file or family JSON file if the user wants to use one that is not bundled with the plugin in its JAR file.

4.2.4 The database

The default database for the plugin is MongoDB, and it stores information to a single collection of objects. Each of these objects contains the gathered data on the features of a single UA. The unique identifier for these objects is the UA string. As mentioned previously: MongoDB has a flexible schema, which means that collections do not enforce the structure of objects stored within it. This means that two objects within the same collection can have a completely different structure and set of fields. An object the plugin stores in the database does have a predefined structure, though, shown as JSON in Code Snippet 11.

The “userAgent” field contains the UA string. The “ua” fields contain the UA data from UA Parser. The “os” fields contain the OS data from UA Parser. The “device” fields contain device data from UA Parser. Lastly the “features” field contains the resulting object from the clients-side tests.

With the advantage of having a database with a flexible schema comes the ability of expanding the object later if new features need to be stored. This can be done without breaking the database schema or having to deal with old data being corrupted or unusable because of schema-mismatch.

```
userAgent : String
uaFamily : String
uaMajor : String
uaMinor: String

osFamily : String
osMajor : String
osMinor : String

deviceFamily : String
deviceIsMobile : Boolean
deviceIsSpider : Boolean

features : { feature : Boolean or Object, … }
```

Code Snippet 11: The database object structure

4.2.5 The Data Access Object

To access the database from the plugin a data access object (DAO) was created. DAO is a design pattern that is meant to separate the business logic and the persistence layer of an application [41]. It consists of an abstract interface that, when implemented, exposes specific methods to the business logic that can be used to read and write data to the database. This can be done without the business logic having to concern itself with the details of the underlying database, making it database-agnostic.

Using the DAO pattern has several advantages, both for the sake of supporting several different database technologies and to enforce separation of concerns within the plugin itself. The business logic in the extensions only interacts with the DAO interface, and as such does not care about what is running underneath to store and retrieve the data. To support this it is necessary to define a domain model as a Java class that represents the objects that should be stored in the database. This allows it to be mapped to the correct database format. Having the domain model defined as a plain old Java object (POJO) means the implementations of the DAO interface can operate on the POJO and let the chosen POJO-to-database mapper do the database-specific work. It is also important to

use a mapper that does not rely heavily on Java annotations, as this could potentially litter the domain model class with implementation-specific annotations that could confuse other mappers that might be needed later.

This pattern also helps in keeping the business logic clean, as all database-specific code is abstracted away behind the DAO object. The DAO object can also be instantiated through the Spring IoC container, moving the database connection and configuration into the context XML, moving this concern out of the code itself.

4.2.5.1 The DAO implementation

The implementation of the DAO was meant for use with our underlying MongoDB database. To be able to store the UA object in MongoDB I needed a mapper that could map from a POJO to the BSON format (Binary JSON) of MongoDB. It was originally decided to use a mapper that was part of the Spring Data initiative, as this would simplify much of the process. The reason I ended up not using the Spring Data mapper was because it would cause severe compatibility issues with the version of Spring that was embedded in Enonic 4.6. I settled on a lightweight alternative called MongoJack, which is a small POJO-to-Mongo library that wraps the classes of the official MongoDB Java Driver and utilizes Jackson, a JSON and XML mapping library.

The MongoDB DAO implementation is instantiated with all the information needed to connect to a MongoDB database and construct a collection object that can be used to create, retrieve, update and delete objects from the database. The collection object is instantiated by a static method supplied by MongoJack, which creates a new collection object, but also takes the type of object that should be persisted as an argument to assist the Jackson mapper.

4.3 The HTTP Interceptor Extension

The first extension needed for the plugin is the HTTP Interceptor extension. Its task is to handle incoming HTTP requests before they are processed by the CMS. The handling consists of checking if the UA making the request has had its featured detected, running both client- and server-side tests if it is an unknown UA. The information gathered can

then be retrieved by the CMS through datasources calling methods in the **FunctionLibrary** extension, as per the second approach mentioned in the Conceptualization section.

4.3.1 Client-side tests

Modernizr handles the client-side tests. What tests are present in each Modernizr file can be customized on the Modernizr website. The default test suite in the plugin contains all available tests from the Modernizr website, this includes tests for all HTML 5 and CSS 3 functionality, as well as miscellaneous Web functionality such as WebGL and Geolocation. All the Modernizr tests are situated in a separate file and can be switched to suit each user by referencing an external Modernizr file in the plugin's properties file.

To send the results of the client-side feature tests to the server, they are put into a cookie with a special ID and format that only uses special characters allowed by RFC 6265 [42]. An alternative that was considered was to send the results back to the server using HTTP POST in an asynchronous call (AJAX), and redirecting once the acknowledgement came back from the server. The reason I ended up going for the cookie-solution because it was less reliant upon getting a timely response from the server. Using AJAX can in some cases end up having the request time out because of connectivity issues or otherwise, especially in the unreliable environment of mobile devices on cellular networks. Using cookies means the result is stored on the client and can be picked up by the server the next time the client is connected and makes a request to the server.

The results of the Modernizr tests are stored in a JSON object on the client, so the intuitive solution for the cookie value would be to use a similar format. Some changes had to be made, though, because it was necessary to replace each delimiter with an RFC 6265 approved character. The reason this is important is because the Java cookie parser strictly adheres to this standard, and will stop parsing a cookie value if it encounters an illegal character.

Colons (:) are not allowed in cookie values, so to split key-value pairs I used “double dash” (--). The outer delimiters of JSON objects are “curly brackets” ({}) and are not allowed in cookie values, they were switched for pipes (|). This is not ideal, as having identical opening and closing delimiters hinders detecting which level of nesting the parser is in. Since the result object only has one level of nesting, this problem was resolved by adding an extra delimiter to denote nested objects, I ended up using “forward slash” (/), because this way the parser on the server can know which level of nesting its in by which character is delimiting each key-value pair.

```
flexbox--1|flexboxlegacy--1|canvas--1|canvastext--1|webgl--1|touch--
0|geolocation--1|postmessage--1|websqldatabase--1|indexeddb--1|hashchange--
1|history--1|draganddrop--1|websockets--1|rgba--1|hsla--1|multiplebgs--
1|backgroundsize--1|borderimage--1|borderradius--1|boxshadow--1|textshadow--
1|opacity--1|cssanimations--1|csscolumns--1|cssgradients--1|cssreflections--
1|csstransforms--1|csstransforms3d--1|csstransitions--1|fontface--
1|generatedcontent--1|video--/ogg--1/h264--1/webm--1|audio--/ogg--1/mp3--1/wav-
-1/m4a--1|localstorage--1|sessionstorage--1|webworkers--1|applicationcache--
1|svg--1|inlinesvg--1|smil--1|svgclippaths--1|input--/autocomplete--
1/autofocus--1/list--1/placeholder--1/max--1/min--1/multiple--1/pattern--
1/required--1/step--1|inputtypes--/search--1/tel--1/url--1/email--1/datetime--
0/date--1/month--1/week--1/time--1/datetime-local--1/number--1/range--1/color--
1
```

Code Snippet 12: An example of the cookie value generated by Detector on Google Chrome.

Once on the server, the cookie is parsed by a method in the Interceptor extension class and converted into a linked hash map that represents all the UA features gotten from the client-side tests.

4.3.2 Server-side tests

The server has access to a small, but potentially useful, set of data about the UA through the UA string in the HTTP request header. To extract this information I used the Java implementation of UA Parser created by Twitter, Inc. [43]. It takes the UA string as an argument and returns an object containing data about the UA family and version, operating system family and version, and the device family, as well as if the device is mobile or a search engine spider.

The underlying parser gets its regular expressions for matching UA strings with their respective UA from YAML files that come with UA Parser. The UA strings are checked

against the YAML files and a best match is found. Each of the categories, UA, OS and Device, is put into the UA domain model object based on the results from UA Parser, along with the results from the client-side tests.

The YAML files that define how to parse the UA strings have to be maintained to be completely accurate. This means that the plugin will not be entirely maintenance-independent as long as it uses such a system. The data gathered from the UA Parser is not necessarily needed, though, and much of the functionality it provides can be implemented in custom client-side tests, which could remove the need to manually update the systems YAML files or similar due to the them being outdated in some way. Due to time constraints I elected to keep the UA Parser for the thesis, but replacing it is something that should be looked into in potential future work.

4.3.3 Intercepting an HTTP request

As mentioned earlier in this chapter: when an HTTP request is made to a server running Enonic CMS with the plugin, the **HttpInterceptor** extension will be invoked before the CMS itself receives the request. It will ascertain if the requesting UA has had its features tested and run the tests if it is a new UA. To check this, the extension asks the DAO object to retrieve a UA object with the requesting UA string from the database. If no object is retrieved, the plugin has encountered a new UA that needs to be tested and stored in the database.

There are two situations in which the requesting UA will not have an entry in the database. The first is if no tests have been run yet, and the second is if tests have been run, but the results have not yet been stored in the database. In the first case the interceptor will generate HTML markup including the Modernizr tests, cookie generation JS and – for the sake of UAs that do not support JS – a noscript element that redirects back to the requested URL. The second case is if the tests have been run, and the interceptor should parse the results and store them in the database.

4.3.3.1 Generating and sending the HTML markup

The HTML markup that is sent to the client is generated as a regular string containing the minimum amount of markup needed for a valid HTML document containing JS. The Modernizr JS is read from the file specified in the plugin's properties, which can be embedded with the plugin's JAR or defined externally by a user. The Modernizr JS and cookie generating JS are both appended to the string inside a "script" element residing in the "head" part of the HTML document. Lastly a "noscript" element is added inside the "body" part of the HTML document. The purpose of this element is to redirect back to the requested URL if the requesting UA does not support JS. Being able to redirect without the help of JS is crucial, as any site using the plugin would be stuck on the test page for any UA lacking JS support or any that have disabled it.

When the generated markup is sent from the server, one of two things will happen: either the UA supports JS and the tests are run normally, a cookie containing the test results is generated and the UA is redirected back to the requested URL. Or the UA does not support JS or has support for it disabled, in which case the "noscript" element will redirect directly back to the requested URL. In this case it will attach a "nojs=true" HTTP GET parameter to the end of the URL. This is to inform the interceptor extension of the lack of JS when it intercepts the redirected request.

4.3.3.2 Intercepting the redirected request

Once the client redirects back to the requested URL, the interceptor will once again query the DAO object for a database entry containing the requesting UA string. Because the test results have not yet been stored, no entry will be found. The interceptor will thus look for the presence of the "nojs" HTTP GET parameter that might be present in the URL, indicating that JS is unsupported or disabled on the client. If it is present the interceptor will add the key-value pair "nojs : true" to the features map that is added to the UA object. This will be the only entry in the features map for this UA object, as no feature tests can be run on the UA as long as JS is unsupported or disabled. Because of the possibility that the requesting UA might support JS, but has it turned off, some kind of mechanism should be in place to handle this special case. For example a timeout value

could be set so that the UA might be tested again later, we will discuss this in a later chapter.

If the HTTP GET parameter has not been set, the interceptor will check for the presence of the cookie that should have been generated by the cookie generating JS that was sent to the UA. If it is present, it means that tests have been run and the test results should be stored in the features map mentioned earlier. The cookie is turned into a linked hash map by the parser mentioned earlier in the "client-side tests" section. Once the client-side test results have been stored, the UA Parser is invoked to extract the information contained in the requesting UA string. The results from the UA Parser are stored in the main fields in the user agent object that is illustrated in the Database section earlier.

Once the client-side and server-side test results have been stored in a UA object the object is passed to the DAO, which maps the object to the format of the underlying database and saves it. Once the object has been saved the job of the interceptor is complete. The data can then be accessed by function library extensions utilizing the DAO.

4.4 The Function Library Extension

For the CMS to access the data gathered by the **HttpInterceptor** extension I created a **FunctionLibrary** extension to work as an API for Enonic datasources to retrieve UA data from the database. The idea was to expose two methods. One method that returns an XML representation of the stored data that users can utilize within the CMS, the other method should simply return a string representing the device family defined by a separate JSON file, which is specified by the user. The second method is supposed to be akin to Dave Olsen's family system that we looked at in chapter 2.

4.4.1 Getting an XML representation of user agent features

Calling the method **getUAFeaturesXML** in the plugin's **FunctionLibrary** extension retrieves an XML representation of the UA object. The reason for needing XML is, as we saw in chapter 3, because Enonic's templates are based on XSLT parsing of XML data

that is provided by its datasources. The UA object in the database is a series of fields containing strings and Booleans, as well as a map containing either a Boolean or a subsequent map. These are simple data structures that can be mapped quite easily by standard POJO-to-XML mappers. For our purposes we used JAXB, which is a part of the standard Java extended library (javax).

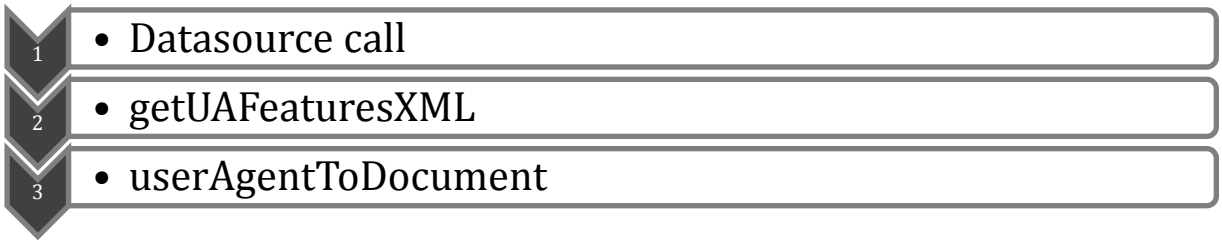


Figure 5: The program flow of a datasource call to getUAFeaturesXML.

When **getUAFeaturesXML** is called from a datasource in Enonic, as seen in Figure 5, it gets the requesting UA from the database and subsequently calls a method that maps the UA object to the XML format using JAXB. The result from the mapper is a string-representation of the resulting XML document. Because Enonic supports JDOM XML Document objects in its datasources, the string representation is converted to an actual Document object by using the JDOM **SAXBuilder** class. The result from the **SAXBuilder** is then returned to **getUAFeaturesXML**, which in turn returns the Document object to the Enonic datasource that invoked it.

4.4.2 Resolving a user agent family

As defined earlier in the thesis: UA families are classifications of UAs that are based on which features they support. In the case of the plugin this means the features the **HttpInterceptor** extension has detected that they support. UA families are defined in a JSON file. One is packaged with the plugin and is used if no other file is defined in an external plugin properties file. If an external JSON file is referenced in a user-specified properties file, that one will take precedence.

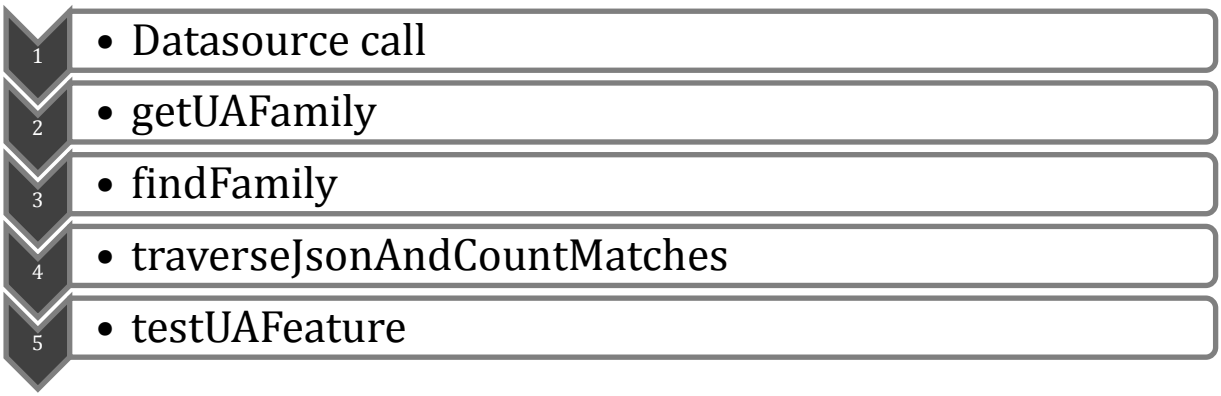


Figure 6: Program flow when a datasource invokes getUAFamily in the function library extension.

To resolve a family for the requesting UA, datasources can invoke the method called **getUAFamily** in the plugin's **FunctionLibrary** extension; the flow of this call is shown in Figure 6. This method will, in short, parse the UA family JSON file and compare its values to that of the UA object to establish a best matching family. The best matching family is defined as the family with the most matching fields and no mismatches compared to the UA object. If no family is found it will fall back to the default family, represented by the string "default". To parse the JSON file and compare the fields to those in the UA object, two methods are used, one to traverse the nested objects of the JSON file and one to actually compare the fields to those of the UA object. These methods are not publicly accessible from datasources as they are meant to be helper functions for **getUAFamily**. To parse the JSON file it is necessary to map the JSON objects into Java objects. This is done by using Jackson, which is the same library used to translate the UA objects into MongoDB objects in the DAO implementation.

Using Jackson it gets a JsonNode object, which includes an iterator that can be utilized to traverse the JSON structure so it can compare the JSON's fields to those in the UA object. Since the top-level fields in the family JSON object are the families themselves, traversing them is done using a simple loop. The internal structure of each family object is more complex and may contain several levels of nesting. This arbitrary number of nested objects suggests that recursion is both the most efficient, and the most practical way to traverse them.

The traversing algorithm, shown in Code Snippet 13, also has to keep track of the number of matches found so that a best matching family can be found, as well as giving

an indicator of a mismatch if it occurs. The algorithm will count the number of matches found, but will return 0 if it encounters a mismatch. The return value from each recursive call is checked, and if it is 0 the caller will also return 0. This means that if any mismatch is found somewhere in the call-stack, the whole call will return 0, giving the algorithm a Boolean-like return value where 0 indicates a mismatch while any positive integer signifies a match of n elements. The return value from the recursive algorithm can thus be used to find not just a match, but the best match by resolving which family has the most matching fields.

```
private int traverseJSONAndCountMatches(JsonNode json, String parent, UserAgent userAgent) {
	int matchedFields = 0;

	Iterator<Entry<String, JsonNode>> jsonIterator = json.fields();
	while (jsonIterator.hasNext()) {
		Entry<String, JsonNode> jsonEntry = jsonIterator.next();
		String key = jsonEntry.getKey();
		JsonNode value = jsonEntry.getValue();

		if (!value.isObject()) {
			if (testUAFeature(key, value.asText(), parent, userAgent)) {
				matchedFields++;
			} else {
				return 0;
			}
		} else {
			int recursivelyMatched = traverseJSONAndCountMatches(value, key, userAgent);
			if (recursivelyMatched == 0) {
				return 0;
			} else {
				matchedFields += recursivelyMatched;
			}
		}
	}
	return matchedFields;
}
```

Code Snippet 13: The recursive method used for calculating the number of matches in a given UA family definition.

Once the algorithm resolves a best matching family, the method traversing the JSON object returns the name of the resulting family as a string. This is then returned to the invoking datasource as an XML element, to make it easy to look up in XSL templates.

4.5 Summary

In this chapter I have described the implementation of my server-side feature detection system as a plugin for Enonic, what technologies were used and why they were chosen. In addition we looked at how the plugin intercepts HTTP requests to do its device and feature detection, how this data is persisted, as well as how I implemented a function library for the CMS to retrieve it.

5 Performance Testing

While the implementation works and can help in detecting the features of requesting UAs, its inherent goal is to improve the performance, and thus the user experience of Web pages across both existing and future devices and UAs. This chapter presents the concept of Mobile Web Performance, as well as how the performance tests were conducted and their results.

5.1 Mobile Web Performance

The performance of Web pages has been steadily increasing along with the constantly improving network infrastructure and computing power of modern desktop computers. With this improved capacity Web developers have been able to create richer experiences on the Web through increased use of resources such as CSS, JS, images and video. Because of this the size of the pages has increased alongside the demand for more powerful computers to render them. With the emergence of mobile computing, Web pages again have to account for being viewed on devices with limited computing power, high network latency and reduced bandwidth. Web pages designed with desktop computers in mind can be orders of magnitude slower on a mobile device, sometimes to such a degree that the Web site is rendered unusable. This is in many ways a strange problem: on one hand mobile devices are much more powerful today than regular computers were ten years ago, but cellular networks are much akin to the regular wired networks in terms of speed and price as they were back then.

The Web technologist, author and speaker Nicholas C. Zakas suggests in his article “The Evolution of Web Development for Mobile Devices” that the problems we face today with Mobile Web performance are rooted in two main concerns: network latency and mobile device hardware limitations [44]. He bases his suggestions on the work of Steve Souders, which we looked at in chapter 1, focusing on his list of best practices when it comes to Web performance [4], especially rule 1 (make fewer HTTP requests) and 11 (avoid redirects).

5.1.1 Network latency

In the context of Web performance we define latency as the delay experienced from sending a request over the network to receiving a response, i.e. the round-trip time of the request. Bandwidth is defined as the amount of data a connection can receive over a specified amount of time, e.g. 20 Mb/s. Bandwidth may be limited by latency.

On regular wired connections, latency over short distances is minimal as the packets sent over the network propagate through physical cables. Barring network congestion, the only source of latency is electrical resistance in the wire material, or the speed of light in the case of optical fiber cables. Because the signals propagate at a finite speed, latency increases with transmission distance, but interference causing transmission loss is kept to a minimum. Desktop computers normally use a wired connection. This means that they will experience minimal latency when sending requests to Web servers, exacerbating the difference between making a request on a desktop computer versus a mobile device connected to a cellular network.

Wireless connections have several sources of interference and general signal loss that may increase latency when making requests to a Web server. Requests made over a wireless network propagate through the air, completely unshielded from any kind of external interference. Radios, microwaves, walls or any other form of electromagnetic or physical barrier may adversely impact the effective bandwidth of a wireless connection, giving wireless networks a much higher potential latency than wired networks.

Cellular networks are especially vulnerable to high latency by nature of their topology. A request from a mobile device has to first go to the nearest cellular tower, and then to a server using the General Packet Radio Service (GPRS) belonging to the mobile service provider, which functions as a gateway to the Internet. This server can then make the actual request to appropriate location (DNS, HTTP or other), the response from which then has to propagate the same way back. Currently these servers are few and centrally located, instead of distributed, giving the user's proximity to them a measurable impact on the latency of a request [45]. Going by Souders' list of principles, rule 1 seems to be quite poignant here, as reducing the amount of requests will directly reduce the latency from the original request until the user sees a fully rendered Web page. Minimizing the

number of redirects (rule 11) is also important in this regard, as each redirection requires a new DNS lookup request and an additional HTTP request. This will naturally increase the total request time substantially.

5.1.2 Mobile device limitations

Modern desktop computers have extremely powerful hardware that have no problems with rendering even the most advanced Web pages. Even though modern mobile devices are quite powerful, especially compared to desktop computers from ten years ago, they do have limited processing power and memory in comparison. In this regard developers have to pay attention to how their Web pages utilize the capabilities of the device they are being viewed on. A Web page that is easily rendered in a desktop browser might cause severe problems or crashes in a mobile browser due to hardware limitations. The two things that are especially important is CPU and memory usage. CPU usage, as well as network access through cellular, Wi-Fi and Bluetooth antennas also impact the battery life of a device, which is already short on modern mobile devices.

Zakas mentions in his article that reducing the amount of JS on a Web page can reduce both the response time and the amount of battery drained as a result of the CPU having to execute less code [44]. The JS engines running on mobile devices are orders of magnitude slower than those in desktop browsers, as we will see later in this chapter. They also vary greatly in performance and can, by themselves, greatly reduce the user experience if a Web page uses a lot of JS. Keeping its use in mobile Web pages to a minimum should thus be a goal when it comes to improving performance.

Another concern Zakas mentions is the limited memory available on mobile devices. Even the latest mobile devices have much less memory than desktop computers, and even less is available for use in Web browsers. While modern desktop computers will never encounter memory issues from loading a single Web page, an excessively memory-intensive Web page might just cause problems on a mobile browser. The biggest reasons for memory issues on a Web page, Zakas claims, are images and hardware accelerated graphics in general [44]. A Web page with a lot of images embedded in its DOM can quickly fill up the memory available to the browser, causing

slowdown and possibly crashes. Modern browsers, including mobile browsers, also hardware accelerate graphics such as images, CSS transitions and animations. This is done because handling graphics on the GPU instead of the CPU will lead to a smoother experience for the user, but it also uses more memory, which is limited on mobile devices. Being aware of these issues and using images and graphics responsibly can go a long way in avoiding potential memory issues on a mobile Web page.

5.1.3 Where the plugin comes in

JS, images, CSS transitions and animations help in creating a rich Web experience for users on desktop computers. Simply cutting them out because they do not work properly on mobile browsers would be unfortunate, and this is where the server-side feature detection system comes in. Being able to detect UA features and tailor the Web pages on the server before sending them to the client could go a long way in allowing developers to get the best of both worlds. Developers can keep their rich Web page for desktops while altering the HTML components for requesting UAs that are known to have limitations such as those mentioned earlier.

Detecting the limitations on the server also assists developers in reducing the amount of requests needed for fully rendering their Web pages when a mobile UA is encountered. JS can be reduced, merged or removed. The resolution of images can be reduced or the number of images can be lowered. CSS files can be merged, or reduced in size to contain only what is absolutely necessary on mobile UAs. Because these concerns are primarily relevant in the context of mobile devices, developers can choose to ignore these issues when developing the regular desktop version, while keeping the capability of tuning the mobile version later.

The plugin could increase the time a request takes to be handled, though. It adds additional database lookups and business logic to the server that it has to execute at least once per HTTP request it receives. To discuss the overall usefulness and effectiveness of the plugin, it is important to figure out just how much of a penalty is incurred on each request when it is active.

5.2 Method

Measuring performance of Web sites is a whole research field in its own right, which was first given a name by Steve Souders back in 2004 [46]. There are many ways of doing it, each aimed at specific parts of request chain or the user experience. Some might target the performance on the back end, while others target the front end exclusively, looking at the execution time of JS and the size of files sent in the response. Others may not look at response or execution times at all, but rather do an analysis of the content of a Web page as it loads to determine the its performance as experienced by the user.

Because the plugin is situated primarily on the back end I focused on that as the common case, but I also had to consider the case where the system encounters an unknown UA and must do tests on the frontend. These two cases are quite different and thus measuring their performance had to be approached differently.

Enonic CMS also has its own device detection system built into it. Since the plugin is meant to replace it I also had a look at it and the plugin comparatively, to establish the performance impact of using the plugin as a replacement. This was done for both cases mentioned above. Even though the plugin detects more features than the built-in system, a severe performance hit might be grounds to argue against using it.

5.2.1 Measuring back-end performance

The performance of the plugin itself can be measured in two ways, either we can measure the time spent by the plugin on the server in its own code using timers, or we can look at the time it takes the browser from making a request until the initial HTML document is received in the response. The most important part is that the environment is kept static between tests, and that the tests reflect normal use cases. It is also important to minimize the possibility of measurement errors due to network irregularities by performing the tests in close succession and on the same network.

I chose to measure the performance of the back end by using the Google Chrome Developer Tools. This allows for looking at the time it takes from a request is sent to the server until a HTML document is received in the response. The Chrome Developer Tools

have a module that illustrates the result of a request on a timeline that shows the individual round-trip times for each resource request to the server, as shown in Figure 7. The initial request is naturally for the HTML document, which is what we are interested in. This shows the amount of time it takes the server to handle the request, look up the UA, handle the UA if it is unknown and send a response.

Figure 7: The Google Chrome Developer Tools Network pane. Showing the demo page making a request as an unknown UA to the server. The blue bars indicate requests for HTML documents. The first one is the redirect from the HttpInterceptor extension to do the Modernizr tests; the second is the actual page.

While the request round-trip time gotten from Chrome is indicative of the actual user experience of latency, we can also look at the processing and rendering time on the server by using Enonic's administration tools. These tools also provide a page trace view after a page has been loaded in the browser. As shown in Figure 9, it displays the page and its portlets as XML, with each element containing attributes for the total rendering time of the page and portlets, their server side processing time as well as the time spent invoking each element's datasource methods. This can be used to see the impact of having the plugin's **FunctionLibrary** methods called, compared to when they are not, giving us data on the performance hit the plugin will inflict on the CMS's page loading times. I used this to establish if there is a significant hit in the performance of a page using the plugin compared to one that only uses Enonic's built in system.

```
▼<page key="0" name="home" display-name="Home" cacheable="true" page-template-name="Destination list" total-time="1361" processing-time="600" run-as-user="admin">
  ▼<functions total-time="40">
     <function name="getMenuBranch" time="15"/>
     <function name="getPreferences" time="5"/>
     <function name="getUAFamily" time="20"/>
   </functions>
  ▼<portlets total-time="721">
    ▼<portlet key="40" name="Banner beach" cacheable="true" total-time="81" processing-time="81" run-as-user="admin">
       <functions total-time="0"/>
     </portlet>
    ▼<portlet key="4" name="Destination list" cacheable="true" total-time="374" processing-time="313" run-as-user="admin">
      ▼<functions total-time="61">
         <function name="getContentBySection" time="61"/>
       </functions>
     </portlet>
    ▼<portlet key="50" name="Portlet welcome" cacheable="true" total-time="173" processing-time="173" run-as-user="admin">
       <functions total-time="0"/>
     </portlet>
    ▼<portlet key="9" name="Event mini list " cacheable="true" total-time="38" processing-time="30" run-as-user="admin">
      ▼<functions total-time="8">
         <function name="getContentBySection" time="8"/>
       </functions>
     </portlet>
    ▼<portlet key="10" name="Portlet photo contest" cacheable="true" total-time="55" processing-time="55" run-as-user="admin">
       <functions total-time="0"/>
     </portlet>
   </portlets>
 </page>
```

Figure 9: A page trace in Enonic. It shows the total rendering time and processing time of a page, as well as a breakdown of the time taken for each portlet and datasource function call.

There are two factors to look at on the back end: the performance of the **HttpInterceptor** extension, and that of the **FunctionLibrary** extension. The performance of the **HttpInterceptor** extension is impacted by whether or not it encounters a new UA string, thus it is important to separate these two cases. The performance of the **FunctionLibrary** extension is wholly reliant on the structure of the UA family definition JSON file, so it needs to be kept static between tests.

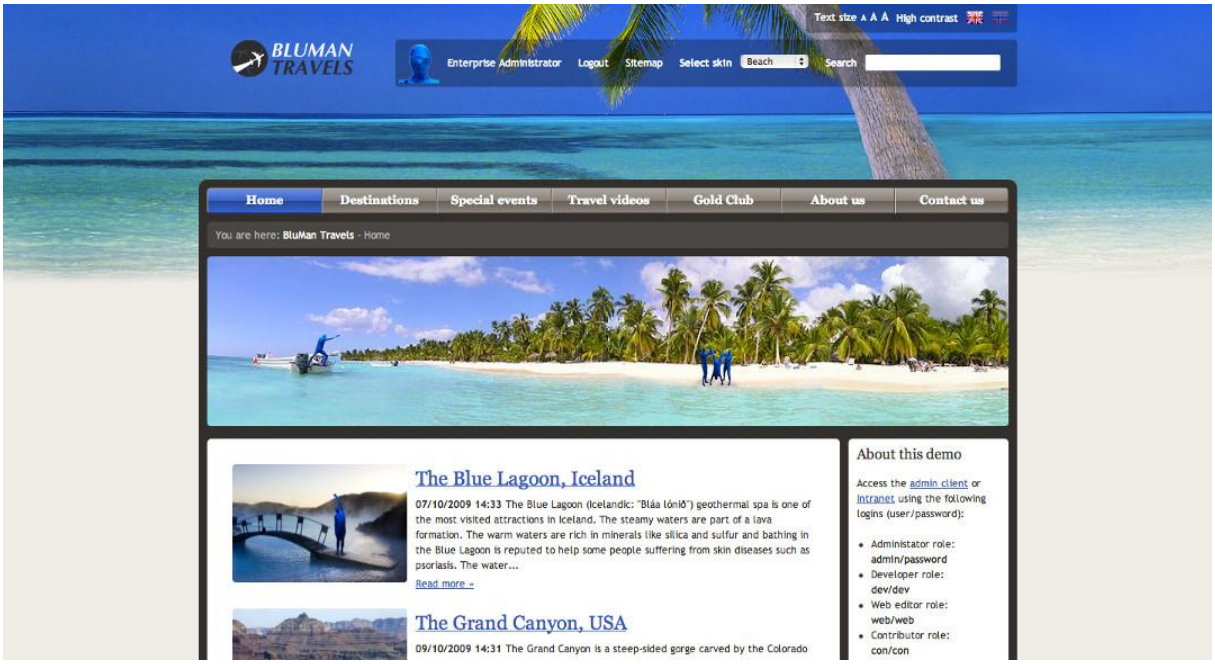


Figure 8: The Bluman Travels test site in Enonic.

Because the results from the **HttpInterceptor** extensions are used by a **FunctionLibrary** extension that is invoked by datasources, it was necessary to add a call to the datasources on he test page. A single datasource should suffice as Enonic

caches the result on a per-session basis, and the **FunctionLibrary** extension will thus not be called twice.

The site we used for testing is the default demo page that is packaged with Enonic 4.6 called Bluman Travels. It is a mock travel site with articles about various tourist locations around the world. It is a good example of a typical web site that uses CMS's like Enonic, and will thus provide a good use-case example. For testing I used the site's front page, as seen in Figure 8, with a datasource call to the **FunctionLibrary** extension method **getUAFamily** in the XSL page template it uses.

5.2.2 Measuring front-end performance

The plugin only ever does anything on the front end if it encounters an unknown UA. When this occurs it sends a response to the client containing some JS code meant to test the features of the requesting UA. To measure the performance on the front end means looking at the time it takes the JS to execute the tests, generate the resulting cookies and reload the page. In the case of no JS support on the UA, the page will simply be reloaded, which is not very relevant in this case, as we are interested in the worst-case scenario where all the JS is executed.

As previously mentioned, different browsers use different JS engines to execute code. These are all implemented differently and thus perform differently. On modern desktop browsers these variations are largely negligible. The JS engines on various mobile devices do have bigger differences in performance, though, and can thus have a significant impact on the request time and resource use on mobile devices. Because the JS based feature tests are core to the functionality of the plugin, I cannot do much to improve the performance of these scripts, lest there arrives a more efficient alternative to using Modernizr. Despite this, it is still interesting to see what kind of performance impact these scripts have on the overall latency of the Web page.

Testing the performance of the JS code running on the client is a matter of measuring the time the code takes to execute. To do this we simply get the time using JS at the start of the actual Modernizr code, and then again after the cookie containing the results has

been generated. This is done using the **getTime** function, which is a part of JS's Date prototype [47]. The result of subtracting the end time with the start time is displayed in an alert box and written down in a spreadsheet. This process is repeated a number of times to get a reasonable dataset from which to calculate an average execution time. I did this in all the major desktop browsers and a few popular mobile browsers to get a comparison between their average execution time.

5.2.3 Comparing the built-in system with the plugin

Enonic has its own device classification system built into it. This system is much simpler than the plugin, but might also be much more efficient when it comes to performance. While the plugin has obvious advantages when it comes to the sheer amount of features it can detect, it has the potential of incurring a larger overhead on each request made to the server, due to the increase in business logic and database queries per request. This poses the question: is the increased overhead worth the additional flexibility provided by the plugin? This is relevant as a big increase in request round-trip time has the potential to negatively impact the user experience of Web sites using the plugin.

Measuring the difference in performance of the two systems can be done using the same method for measuring the performance of the back end. The testing environment should be kept as similar between systems as is practically possible. The references to the result of the Enonic device classification system should be equal in number to the amount of calls datasources make to the plugin's **FunctionLibrary** extension. The difference in performance between the two systems should then present itself in the form of the request round-trip time found in the request timeline in the Chrome Developer Tools. Intuition dictates that the plugin will perform worse in terms of speed, the question is simply: how much worse?

5.3 Results

The following section contains the results from the various performance tests described earlier. The tests were run on an Apache Tomcat webserver running Enonic, and all requests to it were done from the same machine addressing "localhost", except for the JS

performance tests conducted on mobile browsers. The specs of the computer and mobile device are as follows:

The server/computer is a MacBook Pro running OS X 10.8 with a 2.66 GHz Intel Core 2 Duo CPU, 4 GB RAM and a NVIDIA GeForce 9400M GPU.

The mobile device is a Samsung Galaxy S running Android 2.3.3 with a 1 GHz Cortex-A8 CPU, 512 MB RAM and a PowerVR SGX540 GPU.

Both of these are representative of devices among the most widely used in their respective categories around the world at time of writing.

5.3.1 JavaScript performance

I conducted the performance tests of the JS code as mentioned under the "Method" section of this chapter. The results are presented in Figure 10; the three leftmost browsers are desktop browsers, while the three rightmost are mobile browsers. The time represents the average of 40 executions. Testing on mobile browsers was done on a wireless local area network (WLAN) through a Linksys E4200 router. The actual data can be viewed in the appendices.

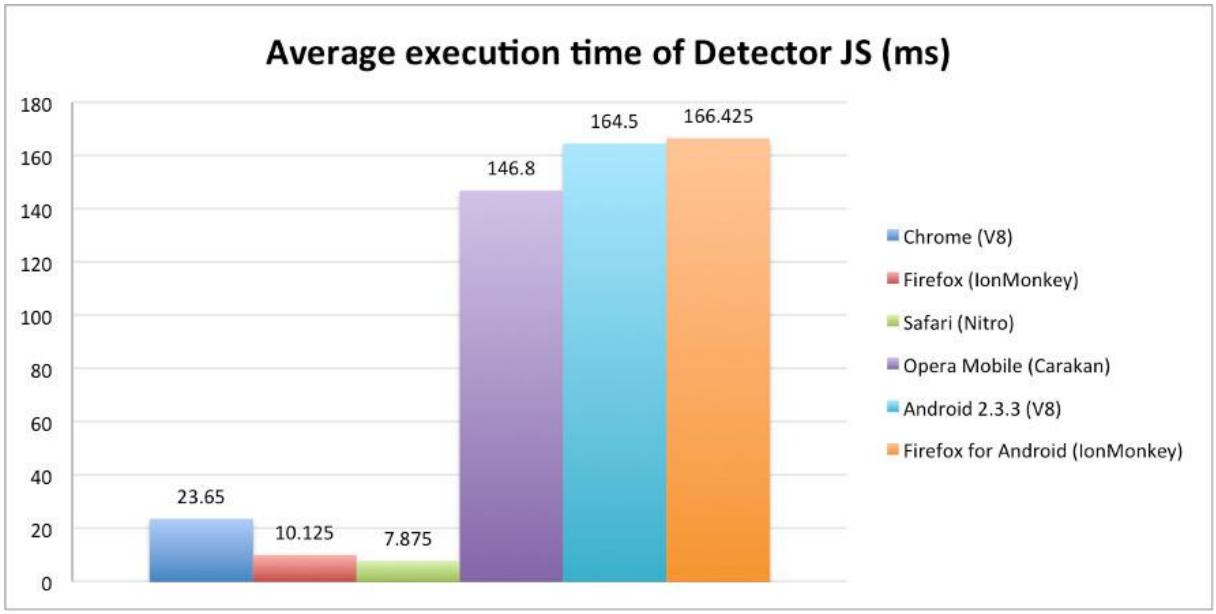


Figure 10: Average execution time in milliseconds of the client-side JS code (Modernizr etc.) on various desktop and mobile browsers. The JS engine used by the browsers is written in parentheses.

There is an obvious difference in the execution time on desktop browsers compared to mobile browsers, which was expected considering the limited system resources on the mobile device. The choice of browsers to test on was simply a matter of choosing three popular browsers among the devices used that also used different JS engines. There are also others, such as Opera for desktops and Chrome for Android, but for the purpose of these tests it was decided that three browsers per device sufficed.

This test was meant to measure the differences in desktop versus mobile JS execution time, and not as a benchmark for comparing engine performance within each individual platform. We will thus not discuss the differences in JS engine performance within the desktop and mobile platforms individually.

5.3.2 Back-end performance

The back-end performance tests were split into two parts, as mentioned under the "Method" section earlier in this chapter. One for measuring the round-trip time of a request made to the test home page, and one for measuring the rendering and processing time on the server itself. Both of these tests were done both with and without the plugin installed, as to measure the impact of using the plugin against the native device classification system.

The request round-trip time also takes into account the extra request that occurs when the interceptor makes an extra redirect to execute the Modernizr tests on the client prior to loading the actual page.

5.3.2.1 Request round-trip time

The round-trip time of a request reflects the time it takes from a user enters a URL to the actual Web site is displayed. In the instance of the plugin it was interesting to see the average round-trip time of requests made with, and without the plugin being installed. For the plugin I measured the average round-trip time for both the worst case and the common case. The worst case is when it encounters an unknown UA and has to gather data, create a UA object and store it in the database. The common case is when it

encounters an existing UA by a single database lookup. The results of the tests are presented in Figure 11, while the actual measurements can be seen in the appendices.

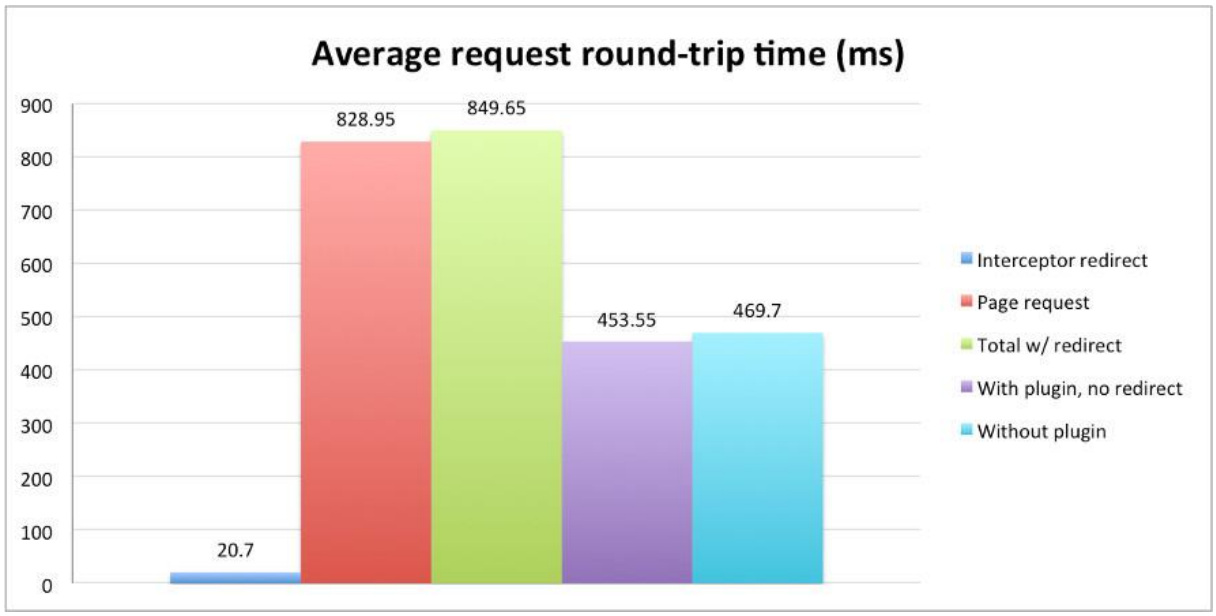


Figure 11: The average round-trip time of requests made using the plugin in both the common, and worst case, along with the round-trip time when not having the plugin installed.

The results suggest that the only time the plugin has a significant impact on the request round-trip time is in the worst case, where it is increased by 81 percent. The differences between the common case and not having the plugin installed are so small that they can be attributed to performance fluctuations on the machine that was used for testing. This test was conducted using Google Chrome, as mentioned earlier, and the interceptor redirect time, which includes the rendering of the client-side JS, is quite close to the average execution time we measured for Chrome's V8 JS engine. The biggest chunk of time spent in the worst case is in the main request, after the redirect. This includes parsing the Modernizr result cookie, instantiating a UA object with the results and marshaling it into a MongoDB document, before storing it in the database. An unknown factor is how long it takes to resolve a UA family when it is invoked through an Enonic datasource. This can be tested using Enonic's built-in page trace tool in its administration interface, which we will look at next.

5.3.2.2 Server processing time

As mentioned earlier, Enonic contains a page trace tool in its administration interface, which gives a breakdown of where the processing and rendering time is spent on the server. This was used this to find the averages in the common case of the plugin, where the UA is known and it only needs to resolve the UA family when the **getUAFamily** method is invoked from a datasource. To test the method I needed a UA family definition file to that **getUAFamily** should use to match the best fitting UA family. The JSON file we used is shown in Code Snippet 14. The actual data can be viewed in the appendices.

```
{
	"chrome" : {
		"uaFamily" : "Chrome",
		"isMobile" : false,
		"features" : {
			"video" : {
				"h264" : true,
				"webm" : true
			}
		}},
	"notmobile" : {
		"isMobile" : false,
		"features" : {
			"flexbox" : true
		}
	},
	"mobile" : {
		"isMobile" : true
	}}
```

Code Snippet 14: The UA family definition JSON used for testing getUAFamily.

From the results shown in Figure 12 it is evident that the time spent executing the **getUAFamily** method is so small as to almost be irrelevant when looking at the processing time, and even more so when looking at the total rendering time. **getUAFamily** takes up 1.9 percent of the average processing time, and 0.7 percent of the average total rendering time. The "Processing-time" includes all other datasource method calls and the creation of the resulting XML data. The "Total-time" includes rendering of the actual HTML markup that is done by filtering the generated XML data through XSLT templates.

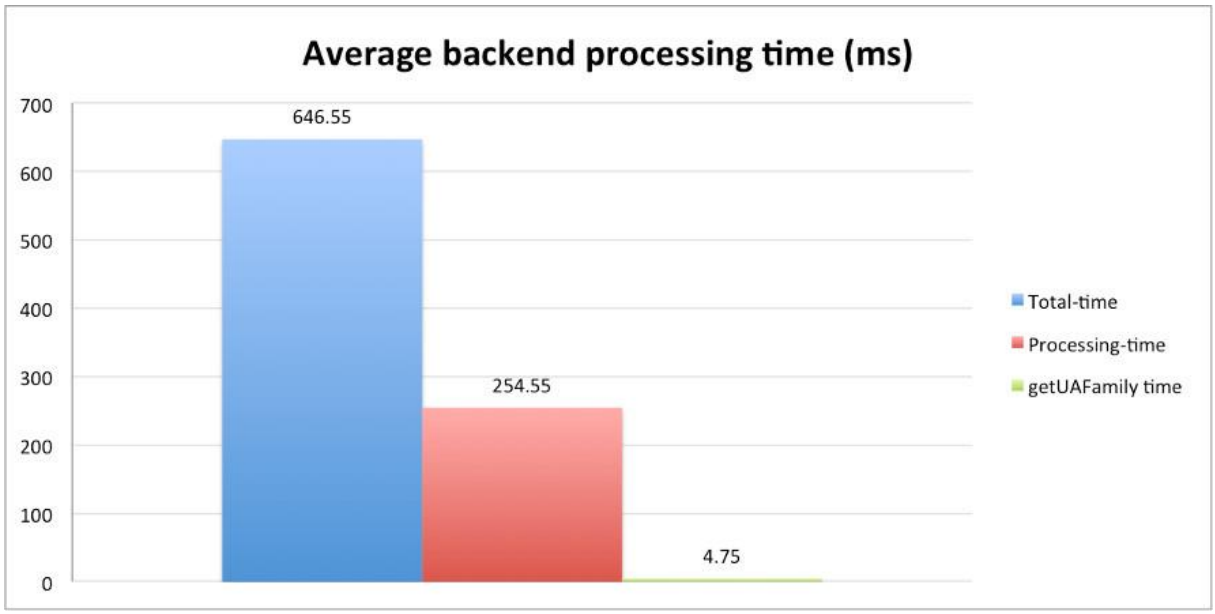


Figure 12: The average backend rendering and processing time spent, as well as the average time spent executing the getUAFamily method.

From these results we can see that most of the performance hit from the plugin is in the case of encountering an unknown UA. In the common case it has a negligible impact on the performance of the request round-trip times and back end processing time. In the common case the plugin does lookups in the database and resolves the UA family each time a datasource invokes **getUAFamily**, which we can see from our results does not impact the performance in any meaningful way. Database queries can be expensive, but they are not in our case when using MongoDB. Its caching scheme uses all available memory on the machine as a cache, and it will thus cache most results, leading to fast database lookups in the common case. MongoDB documents are more similar to objects than relational data, meaning the process of mapping between POJO's and database data is potentially less expensive for MongoDB. It unknown, but likely that using a relational database can increase the time of both **getUAFamily** and the **HttpInterceptor** itself because of the difference in caching schemes and data structures. Though in the grand scheme of things it would be fair to claim that time spent processing data and rendering the HTML, along with network traffic is responsible for the largest portion of time spent, not the database lookups, regardless of the database technology used.

5.4 Summary

In this chapter we have looked at Web performance, what it entails and why it is important in the context of this thesis. I have described the method of how the performance tests of the plugin were conducted, and lastly presented the results of these tests. The results show that the plugin has little to no impact on performance in the common case, while it does impact performance in the case where it has to conduct feature tests on the client. This was expected, though, and seems to not be significant enough to warrant not using the plugin. We will discuss this further in the following chapter.

6 Summary and Conclusions

Throughout this thesis I have presented the theories and concepts that underlie the idea of my server-side feature detection system. In this chapter we will look at the merits of the implementation and discuss its advantages in the context of improving performance of Web sites, with focus on making the Web more Future Friendly [9].

6.1 Why RESS and server-side feature detection?

The growing popularity of using mobile devices to browse the Web has lead to innovations in Web development such as RWD. This concept has quickly become popular among Web developers, especially front-end developers, as it provides a simple way to do feature-detection on the front-end that makes it easy to tailor the style of Web pages to the viewport of each individual browser. The concept does have its problems, though, which has been pointed out by several prominent Web developers [3, 7]. The crux of the concept is that it relies entirely on CSS, and specifically Media Queries, which are situated entirely on the front end. It ends up having the same monolithic markup for every device and also leads to all CSS, scripts and media being sent to every device, as the server is completely oblivious to the capabilities of the requestor. This can lead to performance issues, especially when dealing with larger web pages, scripts and other media. While the layout is changed and elements may be hidden from users on mobile devices by using media queries, all of the content defined in the page's markup is still downloaded by the browser. Large images and scripts that may never be visible to the user will make a page load slower and spend more bandwidth than necessary. Considering how bandwidth is still at a premium in the mobile context, it is unfortunate that this widely adopted development method can be detrimental to the user experience.

To reduce the amount of bandwidth used, lighten the workload on the front end and improve the user experience further, it is necessary to delegate at least some of the responsibility to the back end. Sending the same resources to a mobile device and desktop is counter-productive to what RWD is trying to achieve, namely a future friendly Web experience. Making entirely separate templates for different form-factors

is often called “Device Experiences” because it changes the browsing experience of the page depending on the device. This has been standard fare for many years, and does reduce bandwidth use by limiting the amount of data sent by having it explicitly designed for the requesting device class. These kinds of pages can be made semi-responsive by utilizing certain RWD techniques, such as fluid grids, to adapt to small changes in screen size. They are not responsive in the sense Ethan Marcotte meant when he introduced the concept of RWD, though, and because of their mostly static design are definitely not Future Friendly. Even though RWD has become extremely popular since its introduction in 2010 [26], the report that 82 percent of Alexa’s 100 top sites use server-side detection to tailor some amount of their content [48] lends credence to the idea that using some kind of server-side detection is still useful. This is where the concept of RESS comes in.

As described in chapter 2, RESS suggests combining RWD with specific components of the markup rendered server side to improve bandwidth usage, performance and user experience through optimizations done on the back end. Luke Wroblewski, who came up with the concept, claims that it is effective amongst other things because it allows authors to only create one set of markup with components defined as templates, without having to worry about it working on different devices [7]. This avoids splitting the code base, which would happen in the case of using Device Experiences. The theory surrounding the RESS concept is sound, and should definitely be taken seriously as a stepping-stone towards a solution to creating completely Future Friendly Web sites. While RESS is mostly a theoretical concept, my system implements it practically. It also tackles a few of the problems mentioned by Wroblewski in his original article [7], such as how to define device classes and improving upon the accuracy of device detection by combining server-side UA detection with client-side feature detection.

6.1.1 The Advantage of RESS and server-side device detection

My system, which was inspired by Dave Olsen’s “Detector” project [32], improves server-side detection by including client-side feature detection when encountering unknown UAs, in line with Alex Russel’s suggestion that feature tests should only be run in this case [49]. The addition of these kinds of feature tests means that the server can

be aware of exactly what capabilities the requesting UA has without any prior knowledge. It removes the need to maintain any kind of Device Description Repository (DDR), as the system itself is capable of figuring out the features of new UAs dynamically. This is naturally limited by the feature tests that the system uses, but it is also obvious that updating the tests, which in this case is handled by Modernizr, is a lot less time consuming than updating a central DDR every time a UA is changed to support new features or a brand new type of UA is released. Considering this, the claim can be made that this kind of server-side detection is a step forward in terms of creating a Future Friendly system which is not limited by the data stored in a DDR.

Olsen also suggests a system for handling and easily defining device classes in what he calls "browser families" [10], something that is only mentioned in passing in Wroblewski's article, even though it is key to making a RESS system work [7]. These families are what define the content the server renders in each individual component of a Web page using RESS. This method of defining families is both robust and user friendly both of which are important for a system to enjoy widespread use. It also makes it easy to define new families should the need arise, for example if a new kind of device class is introduced. The modular design of the browser family system allows it to be easily extensible, and thus Future Friendly by adapting alongside the client-side feature tests.

6.1.2 Performance gains with RESS and server-side feature detection

Performance has again become an issue in Web development when considering data traffic over cellular networks (3G, LTE, etc.). Not only is the bandwidth of cellular networks smaller than regular networks, devices also experience much higher latency on them because of routing, among other things [45]. A report published by Juniper Research in April 2013 states that global data traffic on mobile devices will reach 90 000 petabytes a year by 2017. The same report also claims that about 60 percent of this will go through WLAN, not cellular networks, due to network providers building WLAN based "IP-zones" to cope with demand, along with improving their cellular networks [50]. While the widespread usage of WLAN hotspots will lessen burden on cellular networks, 40 percent of traffic will still go through them, which needs to be addressed.

To confront this problem, several issues need to be considered, most of which have to deal with the slow speeds, large distances and prices of data traffic over such networks. On top of this the devices themselves also have limited resources, such as processing power, memory and battery life. To improve performance on the mobile Web, several things need to be addressed: The amount of HTTP requests need to be reduced, images and other media needs to be minimized or eliminated, redirects should be avoided and JS needs to be optimized or its usage should be minimized [44].

RESS can help with all of these issues, as it allows full control of what is delivered to the client after the first request is made. To reduce the amount of HTTP requests, developers can consolidate or reduce CSS and JS where it is needed, as well as reducing the amount of images and other media in the markup, all of which will trigger additional requests to be made to fetch them. This can be done dynamically when certain device classes are encountered. Redirects can be avoided entirely because components are changed on the server, which gives the ability to keep entirely different pages under the same URL. Because of the component-based approach RESS uses, the complexity of dynamically changing between layouts on the server is kept to a minimum. This in turn simplifies many of the steps needed to optimize for weaker devices, such as mobile phones, both through reducing HTTP requests in various ways and by reducing the performance requirements for viewing the site in terms of processing power and memory usage.

The concept of server-side feature detection, along with the browser family system allows my system to adapt to changes and new UAs without relying on DDRs that need to be maintained. The components in RESS give Web pages built using it an inherently modular design, which allows them to be extended to support new families by creating a new family definition and component markup where needed. Olsen also mentions in his article that while these concepts come out of trying to solve mobile issues, they should not be pigeonholed as mobile solutions [10]. RESS and Detector (and similar systems) are not strictly mobile solutions and can help by offering a robust platform for building Future Friendly Web sites and apps, he believes. Considering the merits of these concepts, along with the server-side feature detection and browser family systems, Olsens's claim does not seem too far-fetched.

6.1.3 Disadvantages of using RESS and server-side feature detection

Most of the notable disadvantages of RESS come from the fact that it is so reliant upon device and feature detection. While the concept of delegating the rendering of components to the server and sharing the responsibility of adaptation between both the client and the server is a solid one, device and feature detection has several potential problems that can hamper the functionality of Web site built using RESS. It is reliant on uniquely identifying UAs, which currently can only be reliably done by using the UA string sent with HTTP requests. UA strings, as mentioned in chapter 2, can be spoofed and otherwise misrepresent the actual UA making the request, be it through proxy servers or something else [31]. This might cause the RESS system, such as my plugin, to render components for the wrong device class, or conduct feature tests on a UA that is not the one it makes itself out to be. For instance: certain mobile browsers have settings that allow the user to send the desktop browser version of its UA string. This can, in the case of the my system, lead it to believe the desktop browser only has the features of the mobile version if it is the first time it encounters this UA string. These are edge-cases, but they are possible and have to be sorted out somehow.

Another problem with the feature detection system is that it needs JS to work. JS is commonly activated in modern Web browsers, but users can choose to turn it off. In this case it is impossible to detect features on the client. If a new UA string is encountered and the requesting client has deactivated JS support, the system needs a way to handle the situation so that tests can be run at a later time on a client with JS switched on. This needs to be taken into account to be able to get complete data from every UA, even in the case where certain instances of a UA have JS deactivated.

6.2 Making the Web Future Friendly with RESS

As mentioned in chapter 1, making a Web site Future Friendly consists of several things: thinking “content first”, being data centric, structuring content in a way that is future-ready, as well as robust device and feature detection [9]. Dave Olsen mentions in his article that he believes that RESS with systems such as Detector can create a robust platform for Future Friendly Web sites and apps [10]. While we have seen that the server-side feature detection concept has the potential to provide the latter, the three

former are not something RESS or server-side feature detection enforce explicitly. It can be argued that RESS encourages both “content first” and data-centric mindsets as well, though.

6.2.1 Content first

Thinking “content first” is central to creating Web sites that are Future Friendly. Considering the variety in devices used to view Web sites, it is evident that focusing on the content – the “meat” of the site – is the most important. High-resolution widescreen monitors have allowed desktop focused Web sites to have a lot of extraneous filler outside of the core content because of the extra available screen space. This is naturally a luxury that is not afforded mobile phone screens and other small devices, both current and future. Concepts such as Mobile First [27] can help developers in creating Web sites that focus on the content and other key components, while filtering out all unnecessary containers and filler. Even though RESS should not be seen as a purely mobile concept, it does help enforce many of the same principles that concepts like Mobile First emphasize, such as focusing on the constraints of less capable devices, and designing the site thereafter. RESS helps developers focus on the content of the site first and foremost from being a methodology based around adapting to different devices. The one common thing present on a site, no matter what version a user is viewing, is the content. It is what the user is there for, and it is thus natural to focus on the content and building the site around it. In RESS this is especially poignant, as each component on the site can be changed depending on the device class viewing it, but the content always has to be kept intact somehow. Thus we can argue that RESS as a concept encourages thinking “content first”.

6.2.2 Data centric

Creating sites using RESS can also encourage data-centric approaches, as developers are given incentives to reuse content across components for different device classes. Concepts such as COPE (Create Once, Publish Everywhere) in which content is made independent from the display [51] are natural extensions to the back end of RESS sites. This is because components are meant to adapt and orbit around the content itself to create the best possible user experience, no matter the device. The basic layout of

components in RESS sites means that having data that is interoperable between different components is highly desirable. The concept itself encourages reducing redundant code by reusing components where needed, it would then only seem natural to extend this to reusing data as well, adhering to the principles of COPE, for instance. Being able to reuse data means being data centric from the start, to set up storage of the data in such a way as to optimize the ability to utilize it, no matter the context it is being viewed in.

While RESS itself is not reliant upon a data-centric approach, it can benefit greatly from it. As Luke Wroblewski demonstrates in his blog post describing how all the data in his Bagcheck Web application can be accessed from a command line interface due to it being available through a generalized API [52]. His Bagcheck blog is incidentally also the first site he developed using RESS, and uses as an example in his article describing the concept [7]. Based on this, we can argue that RESS as a concept can both encourage data-centric approaches as well as benefit greatly from them. A Web site built using RESS is highly modular in its design and reliant on being able to adapt its presentation based on the results from its device and feature detection. The ability to have the data be available through a generic API, for instance, would remove one point of uncertainty for developers, as well as making the site itself more Future Friendly.

6.3 Choosing Enonic

CMSs are widely used in larger Web sites that need to generate and handle content, mostly to simplify the administration and usage of the sites. They allow content creators to easily create and publish content on a site without having to know all the underlying technical details. The basis of this thesis was to look into the concepts of Mobile First, RWD and how they tie into the problems developers face with improving both performance and user experience on the mobile Web. The idea was to look into how I could improve performance of sites that are not well adapted to mobile devices. This led me to wanting to look at Web pages within the Norwegian public sector, as these have a tendency to be bloated with content and have layouts specifically made for desktops.

Looking into this I found Enonic, a CMS that is used in several large Web sites in the Norwegian public sector, such as The Norwegian Labor and Welfare Service (NAV), The Norwegian Public Roads Administration (Statens Vegvesen) and The Norwegian Agency for Development Cooperation (NORAD) [35]. Looking at how the system underlying these sites worked, I could establish what is possible with regards to improving their performance on mobile devices and otherwise.

I decided that it would be interesting to look into the merits of the RESS concept, and thus I needed to look into the capability of doing this in Enonic. The CMS has its own system for adapting pages to different devices, but it is very basic and does not provide much in terms of feature detection and accurate definitions of device classes [38]. To be able to use RESS it needed a more accurate detection scheme that allowed it to dynamically detect features on requesting clients and resolve device classes.

It has been noted that many modern CMSs, Enonic included, have attributes that are poorly suited for mobile. This includes things such as not separating text, images and video so that assets can be provided in different sizes, editors that allow content to not be republished everywhere, and HTML templates, JS and CSS that assume desktop bandwidth, CPU and caching behavior [53]. Many of these problems come from the fact that a lot of CMSs conflate content management with Web publishing, which would be entirely at odds with the concept of COPE [51]. This comes back to Sæterås's article that we looked at in chapter 1, where he states that the entire value chain in the Web infrastructure has to be made responsive, not just the front end [6]. I saw RESS as a possible solution to some of these problems, and decided that implementing a server-side feature detection system to improve Enonic's device classification system would be a good way to look into how to improve many of these observed problems with modern CMSs on mobile devices and otherwise.

Enonic's support for plugins, along with its already existing device classification system presented the possibility of making a comparative analysis of my implementation and the native system, giving me a good metric for the possible improvements my system provides. It also meant that it was possible to create such a new detection system for the CMS without hacking the source code and making users reliant on a code base modified

from the original system. The plugin support coupled with its usage in large Web sites within the Norwegian public sector along with its less than ideal device classification system are the main reasons why I chose to use Enonic for my implementation. That is not to say that the concept could not work in, or be relatively easily ported to, other Java-based CMSs, such as dotCMS – a popular open-source web CMS – which also supports plugins built much the same way as Enonic, with OSGi bundles and Spring [54].

6.4 The plugin

As mentioned in chapter 4, the plugin is inspired by Dave Olsen's Detector system as he describes it in his article [10] and from the source code of his Detector project, which is written in PHP [32]. The implementation is meant to contain the necessary features needed to support a RESS solution in the Enonic plugin. This implies that it must have a robust feature detection system along with functionality for defining device classes that can be resolved when needed. There are several differences between Olsen's implementation and mine but they have quite similar functionality, even though the languages, libraries and technologies used are often vastly different. For example: Olsen uses PHP and stores the detected UA features in the file system, while I use Java in Enonic and store data in a NoSQL database. These are underlying technologies, though, and do not change the actual functionality, which for the most part is the same.

6.4.1 Performance impact

The whole point of server-side feature detection systems is to improve the rendering time and general latency of loading Web pages on all devices, as well as increasing the flexibility and accuracy of detecting which device is making a request, and what features it supports. This is all done in the interest of giving developers the tools to create content-centric Web sites without having to worry about creating entirely new Web pages for every device imaginable. This all has the effect of improving rendering time and latency by cutting away all unnecessary fluff from the versions that are sent to less capable device on slow connections, such as mobile phones on cellular networks.

Since the point of the plugin is to improve performance, it was necessary to establish the impact the plugin had on the performance of Enonic. In chapter 5 we went through how I executed the performance tests and their results. I described why performance is important, especially in the mobile context, and how the plugin might impact the performance of Enonic when it is used on a Web site. The results showed that the plugin had little to no impact on the performance of the Enonic demo Web site in the common case, while it did have an impact in the case of having to perform feature tests on the client. In this case there were two culprits: the execution time of the JS, which only showed a significant execution time on mobile browsers, and was largely negligible on desktop browsers, as well as the time spent sending data "over the wire", because doing feature tests requires extra requests to be made.

Based on the results of these tests I claim that the performance impact of using the plugin is negligible compared to not using it. Especially when considering that the only time it showed a significant performance hit was in the rare case of encountering a new UA. In these cases the rendering time of the Web page was increased by 81 percent. This increase in rendering time will ideally only happen to the very first user of that UA that makes a request to the server using the plugin, and only on the first page request. All subsequent requests from that UA will not experience any significant latency compared to an identical server not using the plugin, meaning all users except the first one with a new UA will experience the rendering times of the Web pages without delay.

Based on these results we can conclude that using the plugin will improve the UA feature detection capabilities of Enonic, as well as provide a more flexible device classification system. This will facilitate reductions in loading times by providing the ability to tailor the HTML to each device class, without the plugin impacting the performance of the CMS in any significant way compare to its native device classification system.

6.4.2 The plugin versus Enonic's device classification system

As described previously, Enonic provides its own device classification system [38], which is completely reliant upon analyzing the UA string provided by the requesting UA.

The script that checks the string has to be defined by the administrator of the page itself, and is based on XSLT, a language that is not always easy to write, read, or maintain. The system is based on writing regular expressions (regex) that are matched against the provided UA string, which subsequently outputs a string that is defined along with the matched regex. This string represents the detected device class.

The plugin provides a more detailed detection scheme that not only checks the UA string, but also checks the requesting client for all supported features. In addition it gives administrators the ability to define the device classes based on their detected features and capabilities in a JSON file, which is easy to write, read and maintain, because JSON is designed to be human-readable [55]. The system is also independent of knowing the structure of the UA string, as the client-side feature tests will be able to run no matter what the string looks like, and is thus Future Friendly in a way that the Enonic system cannot be. Being able to detect the features of new, unknown UAs without any manual input from users or administrators can be a big advantage. The device classification system of the plugin is naturally dependent on having a device class definition file that is maintained, but it does not have to be updated every time a new UA comes along. With well thought out class definitions a system can be left unmodified for long periods of time without being in danger of leaving new devices completely unsupported.

6.4.3 Using the plugin in Enonic Web pages

It is not just a matter of installing the plugin to be able to use it. To use it on Enonic Web pages, certain changes need to be made in terms of structure and usage of datasources. Firstly, every page that is made using RESS principles needs to invoke the **getUAFamily** method from the **FunctionLibrary** extension. Upon page load this will resolve a device class and inject it into the XML that Enonic generates before rendering a page template. The injected XML element with the device class string can be checked through conditional statements in the XSLT page templates. This way each block of conditional statements can be considered an individual template for a specified device class, much in the same way Olsen does using Mustache in Detector, as described in chapter 2. This is not entirely dissimilar to how it is done with Enonic's own system. The difference is

that instead of checking the native XML element under “context”, you have to use the custom XML element provided by the datasources through the **FunctionLibrary** extension.

For the plugin to work on every page, a datasource that invokes the **getUAFamily** method has to be defined for every page template, and for every dependent portlet template. Once this is done, the conditional statements within the XSLT template should function as intended. This means that some work has to be done in order to move an existing Web site over to using the plugin, and the amount of work correlates directly to the number of page templates on the given site. The added flexibility provided by the plugin thus comes at a price of increased complexity throughout the site, although the widespread use of datasources in most Enonic Web sites means this tradeoff is not too significant.

6.4.4 Potential problems

There are several potential problems that may be encountered when using the plugin in a production setting. The first one we have already mentioned earlier in this chapter: false or misrepresented UA strings. The plugin has no way of knowing if a UA string attached to a request actually belongs to the requesting UA. It can be changed by the user, the UA itself or by a proxy somewhere on the line between the UA and the server. In these cases the system has to trust that the UA is telling the truth, and store whatever features it detects. There are currently no way to fix this problem outright, as there are no certificates or other way of verifying the validity of a UA string in relation to the UA that sends it.

One way to lessen the problem might be to attach a Time To Live (TTL) on each entry in the database, and rerun the feature tests on the requesting UA of entries that are expired. This approach can lessen the risk of having a misrepresented UA stored in the database forever, but comes with the cost of having to run the tests more frequently. There is also a possibility of being able to verify the UA name and version on the client using additional JS. The navigator global object in JS contains information about the UA it is running in [56]. Doing this is apparently unreliable, though, as the various

attributes, such as appName, returns the wrong name for various UAs. For instance it returns the string “Netscape” for the Gecko- and WebKit-based browsers Firefox and Chrome, respectively [57].

Another potential solution is to test several instances of the same UA string, and discard UA data with outlying results after a set amount of tests have been conducted. This would require a substantial amount of extra business logic in the interceptor extension, such as a scheme for deciding which test results are true and which should be discarded. In the long run this could provide a more robust and trustworthy result compared to the current scheme despite it increasing the number of worst-case requests that trigger a full client-side test suite to be sent.

Another problem that is slightly related to the one mentioned previously, is the case of the UA being tested lacking JS support or having it turned off by the user. It is impossible for the server to distinguish between these two cases, and because JS support is lacking, there is no way to check on the client either. To alleviate the potential problem of redirecting the client to the feature tests on every single request, it is necessary to store some amount of information about the requesting UA. Currently this is done by storing the UA object normally, but with a “nojs” attribute set to “true”. The UA might in fact support JS, though, but have it temporarily disabled by the user. To avoid this false information being stored forever for the UA in question, the database entry might need to have some kind of TTL, like in the case of the fake UA string. Another possible solution may be to store the UA information on a per-session basis in the case of no JS support, having everything be discarded when a session expires.

Several performance tests were performed using the plugin, but they have been done on a small scale on a single server running on a regular MacBook Pro, as stated in chapter 5. There is no way of knowing for sure how the plugin will perform in a production context running on an enterprise-scale server with thousands of requests per day made for hundreds of different pages. In these cases there is no sure way of saying what will happen, and if it will perform as we expect from the performance tests I did. The only way to know this for sure is to actually perform tests on a Web site of this scale.

An example of such a problem could be in the form of caching on a reverse proxy. A reverse proxy is a type of proxy server that gets resources from one or more servers on behalf of a requesting client. The reverse proxy then acts as the origin of the content it gets from the underlying servers [58]. Caching on reverse proxies means that it stores any file that it gets from the underlying infrastructure, and returns it directly upon subsequent requests. Since the plugin manipulates HTML markup based on what class of device is making the request, it is important to disable caching of HTML files in reverse proxies when using RESS systems, as this would lead to the same markup being served all devices, thus negating the effect of using it. Caching of HTML can be kept on a per-session basis, as this would not interfere with the plugin, though this would only improve performance in cases where the cache on the UA itself is either invalidated or turned off entirely.

6.5 Summary

In this chapter we have seen that my server-side feature detection system is a possible step in the right direction towards having a Future Friendly RESS system that can make the Web available, and more importantly tailored to all devices. By looking at concepts such as COPE we have discussed the merits of RESS and how it can provide a platform for a Future Friendly Web that sets content first and orbits around its data, instead of its presentation.

We have looked at the potential problems of RESS and my implementation, which is primarily focused on the uncertainty that is inherent in relying on UA strings for identification. Unfortunately, as we have seen, this is currently the only option available to us. But while this is a potential problem I have made a few suggestions as to how this can be fixed when considering the plugin itself.

I have described why I chose to implement a RESS system generally and why I did it in Enonic specifically. Looking at the relevance of RESS in the context of CMSs is important, especially in the light of arguments that many of the things RESS provides can be used to solve several of the problems we face when using CMSs for mobile devices [53]. We also discussed the functionality of the plugin, how it stands up to the native device

classification system and how existing sites can be converted to using our plugin, and concluded that the extra work of converting to using a system like my plugin can provide a substantial increase in performance and flexibility across devices.

6.6 Conclusion

With this thesis I set out to establish if there existed solutions to the problem of creating responsive, device independent Web sites without having all the workload on the front end. The motivation for this was the emerging dominance of mobile devices being used to browse the Web [1], along with an interest in making Web sites that are Future Friendly [9]. I have presented concepts that have been introduced to alleviate or fix the various problems developers encounter when creating Web sites that need to function on a plethora of UAs and devices, such as RWD [2], Mobile First [27] and RESS [7]. Looking at these I established that RESS might be the best solution, and I presented Dave Olsen's Detector project [10] as a concrete example of how a RESS system could be implemented.

Inspired by Olsen's project I developed a RESS system for the Enonic CMS, which I presented in chapters 3 and 4. Throughout those chapters we looked at the design choices I made and how I used Enonic's Java based plugin support to implement my own RESS system that worked as a plugin for the CMS.

In chapter 5 I demonstrated how the performance tests of the plugin were conducted and their corresponding results. The results were compared to Enonic's own device classification system [38] and I established that the overhead introduced from replacing the native system with my own did not constitute a significant performance impact on the system in terms of page request latency. This means that the performance improvements that the plugin provides as a RESS system will not be reduced by it adding more load on the back end compared to Enonic's own system.

Throughout this chapter we have discussed the various advantages and disadvantages of using RESS in general, and my system in particular. The disadvantages, as stated earlier in this chapter, are mostly rooted in the fact that that system is entirely reliant on

uniquely identifying UAs through the UA strings that are sent along with HTTP requests. UA strings, as we discussed earlier in this chapter, and as stated by others [31], are inherently unreliable and cannot be entirely trusted to represent the actual requesting UA. This is a problem that currently has no complete fix, as it is currently the only standardized method for identifying UAs through HTTP. We have discussed various solutions for alleviating the problem in this chapter, all of which can help in reducing the potential uncertainty that comes with UA strings.

It can be argued that my system adheres to the principles of Future Friendliness [9], and provides these advantages to the Enonic CMS by adding a more accurate feature detection and device classification system. These systems together provide a potential performance boost and added flexibility for developers to create Web pages that adapt to different device classes without the inherent problems present in RWD. In addition to providing the ability to create better performing, responsive pages, the system gives developers the ability to keep a single code base instead of one for every device-specific site.

In the end I claim my implementation of a RESS system for the Enonic CMS is successful in what I set out to do, which was to create a system that allows for creating responsive Web pages without having the front end doing all the work. Barring a few disadvantages the system works as intended and provides the needed functionality of a system supporting RESS. There is always more work that can be done to improve it, such as implementing some of the solutions for alleviating the disadvantages of the system, as well as attempting to make the system independent of the CMS entirely, and have it as a standalone service. The system works as described, though, and can potentially provide performance gains to responsive Web pages made in Enonic today.

6.7 Future work

In its current state the plugin works as intended, but as mentioned earlier it has not been tested in a production environment, and may encounter several problems when subjected to rigorous usage. Future work on the system should include attempting to find stable solutions to these problems, along with actually testing it in a realistic setting.

The next step for the server-side feature detection concept should probably be to make it work independently of superfluous systems, such as CMSs. There are several ways this can be done, the most sustainable and Future Friendly of which is to provide it as an external service through an API, for example using REST principles [59]. Because the system is already written in Java it would not take much work to convert it to run standalone under a regular Java servlet. The function library extension methods could then either keep returning XML, or be changed to return JSON if that is deemed as more practical.

While the lookup part of the application could work just fine as a separate service, the interceptor extension part of it raises several concerns. One of the largest advantages of my system over a regular DDR is that it dynamically detects features of a UA as it encounters it, without the need for human interference. As a third party service this is not as simple because it relies on sending the tests as JS to the requesting Web server. An issue of trust then arises in the sense that the requesting Web server in no way can be sure if the JS it gets from the service is not malicious. The service also has to have systems in place to handle the results from the requesting Web server, as the results sent back could be altered before being forwarded to the service. Having it be a third party API openly available on the Web can thus be deemed to be unsafe in its current form. It could be used in an internal system, though, where the API is situated within the back-end architecture. In this case the server providers would have complete control over what is being sent, and can thus trust what is being sent to and from the server-side feature detection service.

7 Appendices

7.1 Appendix A – JavaScript performance test data

Browser	Chrome (V8)	Firefox (IonMonkey)	Safari (Nitro)	Opera Mobile (Carakan)	Android 2.3.3 (V8)	Firefox for Android (IonMonkey)
Execution time (ms)	26	17	19	355	151	284
	19	11	7	265	171	168
	19	9	8	125	112	138
	22	10	8	39	276	66
	23	19	8	63	65	343
	29	10	9	70	1248	156
	32	10	8	51	96	248
	19	10	6	52	58	66
	23	10	7	55	49	67
	18	10	11	40	354	82
	17	10	10	98	127	196
	35	9	7	46	96	72
	20	9	7	97	100	134
	29	9	7	232	84	104
	18	10	6	143	414	410
	42	10	6	160	447	72
	24	9	7	44	79	75
	21	10	8	37	238	151
	28	9	8	48	222	221
	21	9	10	38	95	79
	25	10	7	48	42	75
	20	9	7	42	53	180
	19	10	7	44	59	131
	20	9	7	44	105	252
	19	9	7	51	111	66
	27	11	7	246	96	70
	24	10	8	104	55	548
	17	10	7	215	243	128
	21	10	7	215	48	124
	30	9	6	233	152	491
	27	10	7	245	81	119
	19	10	7	267	85	91
	37	10	7	388	129	93
	18	10	7	603	71	155
	33	9	8	123	71	70
	21	10	7	98	53	352
	20	10	8	93	44	155
	17	9	8	211	416	84
	19	10	10	120	84	160
	28	10	9	424	100	181
Average	**23.65**	**10.125**	**7.875**	**146.8**	**164.5**	**166.425**

7.2 Appendix B – Request round-trip time test data

	Interceptor redirect	Page request	Total w/ redirect	With plugin, no redirect	Without plugin
Time (ms)	35	2026	2061	604	888
	11	1205	1216	504	614
	12	1002	1014	457	463
	11	1091	1102	501	437
	14	1020	1034	421	563
	10	879	889	441	503
	11	992	1003	414	516
	11	645	656	466	432
	10	636	646	455	358
	171	620	791	420	484
	12	768	780	439	454
	8	736	744	424	438
	18	635	653	598	428
	11	554	565	403	399
	18	1165	1183	501	382
	10	552	562	422	421
	12	528	540	409	421
	11	484	495	369	348
	8	503	511	386	384
	10	538	548	437	461
Average	**20.7**	**828.95**	**849.65**	**453.55**	**469.7**

7.3 Appendix C – Back-end performance test data

	Total-time	Processing-time	getUAFamily-time
Time (ms)	2046	866	7
	1012	366	5
	733	305	12
	667	236	4
	953	261	5
	687	247	7
	525	225	4
	725	339	4
	494	180	6
	472	190	4
	465	177	3
	480	200	4
	647	256	5
	436	179	4
	476	205	3
	438	168	4
	428	180	3
	451	180	3
	383	155	4
	413	176	4
Average	**646.55**	**254.55**	**4.75**

8 Bibliography

[1] M. Murphy and M. Meeker, "Top mobile internet trends," *KPCB Relationship Capital,* 2011.

[2] E. Marcotte, *Responsive Web Design*: A Book Apart, 2011.

[3] J. Grigsby. (2010, August 12, 2012). *CSS Media Query for Mobile is Fool's Gold*. Available: http://blog.cloudfour.com/css-media-query-for-mobile-is-fools-gold/

[4] S. Souders, "High-performance web sites," *Commun. ACM,* vol. 51, pp. 36-41, 2008.

[5] K. Matsudaira, "Making the mobile web faster," *Commun. ACM,* vol. 56, pp. 56-61, 2013.

[6] J. A. Sæterås. (2011, September 12, 2012). *Next steps of Responsive Web Design*. Available: http://mpulp.mobi/2011/05/next-steps-of-responsive-web-design/

[7] L. Wroblewski. (2011, August 15, 2012). *RESS: Responsive Design + Server Side Components*. Available: http://www.lukew.com/ff/entry.asp?1392

[8] L. Wroblewski. (2011, September 11, 2012). *Future Friendly*. Available: http://www.lukew.com/ff/entry.asp?1407

[9] L. J. Wroblewski, Scott; Frost, Brad; Keith, Jeremy; Gardner, Lyza D.; Jehl, Scott; Rieger, Stephanie; Grigsby, Jason; Rieger, Bryan; Clark, Josh; Kadlec, Tim; Leroux, Brian; Trasatti, Andrea. (2013, April 2, 2013). *Future Friendly*. Available: http://futurefriend.ly/

[10] D. Olsen. (2012, August 10, 2012). *RESS, Server-Side Feature-Detection and the Evolution of Responsive Web Design*. Available: http://www.dmolsen.com/mobile-in-higher-ed/2012/02/21/ress-and-the-evolution-of-responsive-web-design/

[11] T. Berners-Lee. (1989, April 3, 2013). *Information Management: A Proposal*. Available: http://www.w3.org/History/1989/proposal.html

[12] W3C. (1992, April 3, 2013). *HTML Tags*. Available: http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html

[13] ISO, "8879: 1986, Information processing-Text and office systems-Standard Generalized Markup Language (SGML)," *ISO, Geneva,* 1986.

[14] H. W. Lie and B. Bos, *Cascading style sheets: Designing for the web*: Addison-Wesley Professional, 2005.

[15] W3C. (2011, April 2, 2013). *Cascading Style Sheets (CSS) Snapshot 2010*. Available: http://www.w3.org/TR/CSS/

[16] J. Kyrnin. (2013, April 22, 2013). *Origin of the Em*. Available: http://webdesign.about.com/od/fonts/qt/em-origins.htm

[17] W3C. (2011, March 22, 2013). *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification* Available: http://www.w3.org/TR/CSS21/

[18] T. Berners-Lee. (1991, March 15, 2013). *World Wide Web*. Available: http://info.cern.ch/hypertext/WWW/TheProject.html

[19] W3C. (2012, April 5, 2013). *A Short History of JavaScript*. Available: http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[20] R. Sharp. (2010, February 20, 2013). *What is a Polyfill?* Available: http://remysharp.com/2010/10/08/what-is-a-polyfill/

[21] Sun. (1997, March 10, 2013). *The Java Language Environment*. Available: http://www.oracle.com/technetwork/java/intro-141325.html

[22] Oracle. (2013). *The History of Java Technology*. Available: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html
[23] TIOBE. (2013, April 21, 2013). *TIOBE Programming Community Index*. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[24] R. Johnson, *Expert one-on-one J2EE design and development*: Wrox, 2004.
[25] S. Champeon. (2003, February 25, 2013). *Progressive Enhancement and the Future of Web Design*. Available: http://www.hesketh.com/thought-leadership/our-publications/progressive-enhancement-and-future-web-design
[26] E. Marcotte. (2010, May 10, 2012). *Responsive Web Design*. Available: http://www.alistapart.com/articles/responsive-web-design/
[27] L. Wroblewski, *Mobile First*: A Book Apart, 2011.
[28] W3C. (2006, January 18, 2013). *Device Description Repository Requirements 1.0*. Available: http://www.w3.org/TR/2006/WD-DDR-requirements-20060410/
[29] ScientiaMobile. (2012, January 10, 2013). *What is WURFL?* Available: http://wurfl.sourceforge.net/wurfl.php
[30] W3C. (2008, January 18, 2013). *Device Description Repository Simple API*. Available: http://www.w3.org/TR/2008/REC-DDR-Simple-API-20081205/
[31] C. Krycho. (2012, August 5, 2012). *User Agent Detection Will Get You In Trouble* Available: http://www.chriskrycho.com/web/posts/user-agent-detection-will-get-you-in-trouble/
[32] D. Olsen, "Detector," 0.8.5 ed. GitHub, 2011.
[33] D. Olsen. (2012, April 5, 2013). *Detector v0.8.5 Released: Now Being Used on* http://www.wvu.edu. Available: http://dmolsen.com/2012/02/21/ress-and-the-evolution-of-responsive-web-design/
[34] Enonic. (2013, March 10, 2013). *Enonic CMS - Step into control*. Available: https://enonic.com/en/home/enonic-cms
[35] Enonic. (2013, March 10, 2013). *Listen to our customers*. Available: https://enonic.com/en/home/reference-customers
[36] Enonic. (2013, March 10, 2013). *Enonic CMS product comparison*. Available: https://enonic.com/en/home/enonic-cms/product-comparison
[37] Enonic. (2013, March 10, 2013). *Datasources*. Available: https://enonic.com/en/docs/enonic-cms-47?page=Datasources
[38] Enonic. (2013, March 10, 2013). *Device Classification*. Available: https://enonic.com/en/docs/enonic-cms-47?page=Device+Classification
[39] Enonic. (2013, March 10, 2013). *Plugin Overview*. Available: https://enonic.com/en/docs/enonic-cms-47?page=Plugin+Overview
[40] 10gen. (2013, February 17, 2013). *Data Modeling Considerations for MongoDB Applications*. Available: http://docs.mongodb.org/manual/core/data-modeling/
[41] Oracle. (2002, February 12, 2013). *Core J2EE Patterns - Data Access Object*. Available: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html
[42] A. Barth. (2011, February 20, 2013). *HTTP State Management Mechanism*. Available: http://tools.ietf.org/html/rfc6265
[43] S. Jiang, "ua_parser Java Library," ed. GitHub, 2013.
[44] N. C. Zakas, "The Evolution of Web Development for Mobile Devices," *ACM Queue,* vol. 11, pp. 30-39, 2013.

[45] I. Nir. (2012, April 10, 2013). *Latency in Mobile Networks – The Missing Link*. Available: http://calendar.perfplanet.com/2012/latency-in-mobile-networks-the-missing-link/
[46] S. Souders. (2010, April 6, 2013). *WPO – Web Performance Optimization*. Available: http://www.stevesouders.com/blog/2010/05/07/wpo-web-performance-optimization/
[47] E. ECMAScript and E. C. M. Association. (2011, March 25, 2013). *ECMAScript Language Specification*. Available: http://www.ecma-international.org/ecma-262/5.1/
[48] R. Cremin. (2012, April 12, 2013). *Server-Side Mobile Web Detection Used by 82% of Alexa Top 100 Sites*. Available: http://www.circleid.com/posts/20120111_analysis_of_server_side_mobile_web_detection/
[49] A. Russel. (2011, August 15, 2013). *Cutting The Interrogation Short*. Available: http://infrequently.org/2011/01/cutting-the-interrogation-short/
[50] N. Bhas. (2013). *Mobile Data Offload & Onload - Wi-Fi, Small Cell & Carrier-Grade Strategies 2013-2017*. Available: http://www.juniperresearch.com/reports/mobile_data_offload_&_onload
[51] D. Jacobson. (2009, October 8, 2012). *COPE: Create Once, Publish Everywhere*. Available: http://blog.programmableweb.com/2009/10/13/cope-create-once-publish-everywhere/
[52] L. Wroblewski. (2011, April 20, 2013). *Bagchecking in the Command Line*. Available: https://bagcheck.com/blog/8-bagchecking-in-the-command-line
[53] J. Grigsby. (2012, October 20, 2012). *Our Content Management Systems are the Mainframes of the Mobile Era*. Available: http://blog.cloudfour.com/our-content-management-systems-are-the-mainframes-of-the-mobile-era/
[54] dotCMS. (2013, April 18, 2013). *dotCMS Home Page*. Available: http://dotcms.com/
[55] D. Crockford. (2009, April 25, 2013). *Introducing JSON*. Available: http://json.org/
[56] Mozilla. (2013, April 20, 2013). *window.navigator*. Available: https://developer.mozilla.org/en-US/docs/DOM/window.navigator
[57] Mozilla. (2012, April 20, 2013). *window.navigator.appName*. Available: https://developer.mozilla.org/en-US/docs/DOM/window.navigator.appName
[58] Apache. (2011, April 20, 2013). *Forward and Reverse Proxies*. Available: http://httpd.apache.org/docs/2.0/mod/mod_proxy.html - forwardreverse
[59] R. Thomas, "Architectural Styles and the Design of Network-based Software Architectures - Chapter 5," *Irvine: University of California,* pp. 76-105, 2000.